Modellpartitionierung in JAMES II

Eine Studienarbeit von Roland Ewald
Betreuer: Dipl.-Inf. Jan Himmelspach

2. September 2008

Inhaltsverzeichnis

1 Einleitung **3**
- 1.1 Warum Partitionierung? . . . 3
- 1.2 Formulierung und Einordnung des Problems . . . 4
- 1.3 Begriffsklärung: DEVS . . . 8
- 1.4 Terminologie . . . 9
- 1.5 Verwendete Formatierungen . . . 10

2 Vorhandene Ansätze **11**
- 2.1 Überblick . . . 11
- 2.2 Das KL - Verfahren . . . 13
- 2.3 Rekursive spektrale Bisektion . . . 15
- 2.4 Genetische Algorithmen . . . 18
- 2.5 DEVS - Partitionierung mit Kostenfunktionen . . . 23
- 2.6 Ausnutzung von Symmetrien . . . 26
- 2.7 Angrenzende Forschungsgebiete . . . 31
- 2.8 Zusammenfassung . . . 32

3 Vorüberlegungen zur Umsetzung **34**
- 3.1 Allgemeines zu den Vorüberlegungen . . . 34
- 3.2 Anforderungsanalyse . . . 35
  - 3.2.1 Anforderungen an einen Partitionierungsalgorithmus . . . 35
  - 3.2.2 Anforderungen an eine Komponente zur Modellpartitionierung . . . 39
- 3.3 Abgrenzung der Aufgabe . . . 40
- 3.4 Entwicklung der Framework-Architektur . . . 41
  - 3.4.1 Entwurf eines geeigneten Gesamtkonzepts . . . 41
  - 3.4.2 Attributierung von Modellelementen . . . 44
  - 3.4.3 Definition von Constraints . . . 47
  - 3.4.4 Erweiterbarkeit hinsichtlich Load Balancing . . . 48
- 3.5 Erarbeitung von Testkriterien . . . 49

4 Details zur Umsetzung **51**
- 4.1 Implementierte Algorithmen . . . 51
- 4.2 Repräsentation von Graphen . . . 53
- 4.3 Testverfahren und Darstellung der Ergebnisse . . . 55
- 4.4 Schnittstelle zum Restsystem . . . 56

5 Ein Algorithmus für DEVS-Modelle **58**
5.1 Genereller Ablauf 58
5.2 Grundidee zur Partitionierung 60
5.3 Top-down - Phase 61
5.4 Entscheidungsphase 65
5.4.1 Runde 1: Ausweitung der Blöcke mit bereits zugeordneten Knoten 66
5.4.2 Runde 2: Wahl der Prozessoren gemäß Beschränkungen in den höheren Ebenen 67
5.4.3 Runde 3: Wahl der restlichen Prozessoren 74
5.4.4 Ende der Entscheidungsphase 74
5.5 Bottom-up - Phase 74
5.6 Fazit 76

6 Analyse der Testergebnisse **79**
6.1 Analyse der Parameter des DEVS-Algorithmus 80
6.2 Testegebnisse des DEVS-Algorithmus 82
6.3 Testegebnisse des Standardalgorithmus 85

7 Zusammenfassung **88**
7.1 Auswertung 88
7.2 Ausblick 89

A UML-Diagramme der Komponenten **92**

B Flussdiagramme des DEVS - Algorithmus **95**

C Weitere Tests des DEVS-Algorithmus **101**

Abbildungsverzeichnis **114**

Tabellenverzeichnis **116**

Literaturverzeichnis **117**

Kapitel 1

Einleitung

Verwandle große Schwierigkeiten in kleine und kleine in gar keine.
Chinesisches Sprichwort

1.1 Warum Partitionierung?

Im Jahr 2000 begann die Universität Stanford mit der Initiative „Folding at home“ [32], einer groß angelegten, Internet-basierten Simulation der Proteinfaltung. Proteinfaltung ist ein hoch komplexer chemischer Prozess, bei dem sich die Aminosäuren eines Eiweißes nach und während dessen Synthese räumlich anordnen. Eine genaue Simulation der dabei auftretenden physikalischen und chemischen Phänomene würde selbst heutige Supercomputer mehrere Jahre beschäftigen [23]. Seit 1999 investierte IBM mehr als 100 Millionen US-Dollar, um die Entwicklung der Supercomputerarchitektur *BlueGene* [33] zu ermöglichen, die ebenfalls dieses Problem adressiert.

Ähnlich rechenaufwändige Anwendungsbereiche für Simulationen lassen sich schnell finden, sei es die Simulation von VLSI-Schaltkreisen, Robotern oder meteorologischen Phänomenen - es besteht also kein Zweifel über den Bedarf an leistungsfähigen Simulationen, sowohl in ökonomischer als auch in wissenschaftlicher Hinsicht.

Wie kann dieser Bedarf in naher Zukunft gestillt werden? Eine Möglichkeit bietet sicherlich die Entwicklung neuer und leistungsfähigerer Computer, dies ist jedoch extrem kostspielig und langwierig.

Ein anderer Ansatz zur Beschleunigung von Simulationen liegt daher in der Aufteilung der Aufgabe auf verschiedene Computer, die in diesem Zusammenhang auch verallgemeinernd als *Prozessoren* bezeichnet werden.

Besitzt das zu simulierende Modell Elemente stochastischer Natur, ist eine verteilte Ausführung meist recht einfach: Da stochastische Simulationen mehrfach ausgeführt werden müssen um signifikante Informationen zu gewinnen, kann die Simulation gleichzeitig auf mehreren Prozessoren ausgeführt werden. Diese Art der verteilten Simulation hat den Vorteil, dass die Prozessoren untereinander nicht kommunizieren müssen, weil sie unabhängige Simulationsläufe berechnen.

Leider ist diese Vorgehensweise nicht bei allen Anwendungen möglich, so dass eine Verteilung auf verschiedene Computer in vielen Fällen nur durch das Auf-

teilen des zu simulierenden Modells erreicht werden kann. Die Aufteilung eines Modells, auch als *Partitionierung* bezeichnet, ist eine wichtige Voraussetzung zur effizienten verteilten Ausführung von Simulationen:

Angenommen, der komplexeste Teil eines Modells wird dem Langsamsten der verfügbaren Prozessoren zugewiesen. Auf Grund der Tatsache, dass alle Prozessoren nun zusammen an *einem* Simulationslauf rechnen, werden die anderen Prozessoren zu bestimmten Zeitpunkten Teilergebnisse des langsamsten Prozessors benötigen. Daraus ergeben sich für sie lange Wartezeiten, und die Simulation wird im schlimmsten Fall um Größenordnungen langsamer ausgeführt als dies nötig wäre.

Es ist also besonders wichtig, dass das Modell vor der verteilten Simulation so „gerecht“ wie möglich auf die vorhandenen Computer aufgeteilt wird. Dieses Phänomen, also die Abhängigkeit der Gesamtleistung eines System vom jeweils „schwächsten“ Element, ist in der Biologie auch als das *Liebigsche Gesetz des Minimums* bekannt und stellt ein zentrales Problem bei der Entwicklung verteilter Systeme dar.

Diese Studienarbeit beschäftigt sich mit der *initialen* Partitionierung von Modellen, womit die Zerlegung und Verteilung eines Modells auf die vorhandenen Prozessoren *vor* der Ausführung der Simulation gemeint ist. Im Kontrast dazu gibt es außerdem die Methode des *Load Balancing*, bei der die Rechenlast der Prozessoren zur Laufzeit angepasst wird.

Im Idealfall greifen beide Mechanismen bei einer verteilten Simulation ineinander: Zunächst wird das Modell so gut wie möglich partitioniert, damit wenig Load Balancing nötig ist. Danach wird die Simulation gestartet und zu gewissen Zeitpunkten ein Load Balancing durchgeführt, damit die dynamische Entwicklung der Rechenlast berücksichtigt werden kann.

Um Komponenten zur Modellpartitionierung in das Simulationssystem JAMES II zu integrieren, werden in den Kapiteln 3 und 4 Anforderungen an mögliche Algorithmen diskutiert und eine Architektur entwickelt, die ihre Implementierung erleichtern soll.

Zuvor jedoch werden in Kapitel 2 verschiedene bekannte Partitionierungsverfahren diskutiert. Es wird aufgezeigt, auf welche Weise das Problem formuliert werden kann und wo dabei Berührungspunkte zu anderen wissenschaftlichen Themengebieten liegen.

Im Verlauf der Arbeit wird argumentiert, dass Partitionierungsalgorithmen für Modelle die Eigenschaften der zugrunde liegenden Formalismen so weit wie möglich ausnutzen sollten. Kapitel 5 enthält zur Illustration dieser These den Entwurf eines exemplarischen Partitionierungsalgorithmus , der auch die in den vorigen Kapiteln erarbeiteten Anforderungen erfüllt.

Im letzten Teil der Arbeit wird die Leistungsfähigkeit der entwickelten Komponenten geprüft (Kapitel 6) und die Ergebnisse der Arbeit werden zusammengefasst (Kapitel 7).

1.2 Formulierung und Einordnung des Problems

Um ein Modell zu partitionieren, also zu zerlegen, muss man seine Struktur kennen. Dabei sind zur Lösung dieses Problems vorerst nur drei Informationen über das Modell von Bedeutung:

1. Die Elemente, aus denen das Modell besteht

2. Der Rechenaufwand für die Simulation eines einzelnen Elements

3. Die Menge der Daten, die zwischen den Elementen während der Berechnung ausgetauscht werden muss

Für die Darstellung dieser Informationen bietet sich ein gewichteter Graph an, bei dem man die Elemente des Modells als Knoten repräsentiert, deren Berechnungsaufwand als Knotengewichte und die Kommunikation zwischen ihnen als Kanten, die ebenfalls ein Gewicht besitzen.

In diesem abstrakten Modell des eigentlichen Modells sind alle Informationen gespeichert, die für dessen Partitionierung bedeutsam sein können.

Des Weiteren sind Graphen wohldefinierte mathematische Objekte, deren Eigenschaften bereits relativ gut bekannt sind. Da man zu partitionierende Modelle also zunächst als Graphen betrachten kann, soll hier das in der theoretischen Informatik bekannte Problem der Graphenpartitionierung [1] vorgestellt werden:

Sei ein Graph $G = (V, E)$ mit gewichteten Kanten und Knoten gegeben, sowie eine natürliche Zahl p. Gesucht sind p paarweise disjunkte Teilmengen $V_1, V_2, \ldots, V_p$ von V, für die gilt:

1. $\bigcup_{i=1}^{p} V_i = V$

2. $V_i \approx \frac{W}{p}$ für i = 1, ..., p, wobei W_i bzw. W die Summe der Gewichte aller Knoten in V_i bzw. in V bezeichnet

3. Die *Schnittgröße*, das heißt die Summe der Gewichte aller Kanten zwischen Knoten aus verschiedenen Teilmengen, ist minimal

Die Teilmengen $V_1, V_2, \ldots, V_p$ werden im Folgenden auch als *Partitionsblöcke* oder schlicht *Blöcke* bezeichnet.

Dieses Problem hat sich als NP-hart herausgestellt [2], und selbst für $p = 2$ ist es noch NP-vollständig, wenn man fordert, dass beide Blöcke gleich groß sein sollen [28]. Das bedeutet, dass es - im höchstwahrscheinlichen Fall von $P \neq NP$ - keinen Algorithmus gibt, der dieses Problem effizient auf einer deterministischen Turingmaschine lösen kann. Effiziente Algorithmen besitzen eine Zeitkomplexität, die in $O(n^k)$ liegt, und deterministische Turingmaschinen bilden das theoretische Modell aller bis heute realisierbaren Rechenmaschinen.

Für die Partitionierung von kleineren Graphen mag die Entwicklung eines optimalen Algorithmus mit exponentieller Laufzeit noch zweckmäßig sein. Jedoch haben die zu partitionierenden Modelle oftmals mehrere tausend atomare Teilkomponenten. Damit ist klar, dass die Entwicklung eines praktisch einsetzbaren Algorithmus, der optimale Ergebnisse liefert, vorerst nicht möglich ist.

In dieser Arbeit geht es also darum, *annähernd* optimale Algorithmen vorzustellen und zu entwickeln. Sie haben den entscheidenden Vorteil, dass sie noch effizient ausgeführt werden können.

Interessant dabei ist, dass man das Problem auf Grund dieses Sachverhalts auch anders definieren kann: Gesucht sind die paarweise disjunkten Teilmengen $N_1, \ldots, N_p$ von N mit $\bigcup_{i=1}^{p} N_i = N$, für welche die Funktion

$$partition_{cost}(N_1, \ldots, N_p) = \sum_{i=1}^{p} (|W_p/p - W_i|) + cut(N_1, \ldots, N_p)$$

ein Minimum annimmt. Dabei sei *cut* eine Funktion, welche die Anzahl der Kanten zwischen den angegebenen Knotenteilmengen berechnet. Die Funktion wird aber nicht allein von der Schnittgröße beeinflusst, auch die Summe aller Abweichungen von der durchschnittlichen Anzahl der Knoten geht in die Berechnung mit ein.

Damit kann das Finden einer Partition mit möglichst gleichmäßig großen Partitionsblöcken und einer dabei möglichst geringen Schnittgröße auch als Optimierungsproblem definiert werden, bei dem ein Minimum der Funktion $partition_{cost}$ gefunden werden muss. Natürlich kann diese Funktion noch weitere für die Qualität einer Partition ausschlaggebende Faktoren aufnehmen, oder den Sachverhalt gänzlich anders definieren. Das Beispiel soll nur illustrieren, wie einfach sich auf dieses Problem ein weiteres Feld von bereits entwickelten Algorithmen und Heuristiken anwenden lässt, denn die lineare Optimierung ist ein großes Teilgebiet der diskreten Mathematik.

Die Herausforderung bei der Partitionierung von Modellen liegt aber nicht nur in der Komplexität der eigentlichen Aufgabe, es wird bereits schwierig, die benötigten Informationen zur Partitionierung zu erfassen: Während die Ermittlung der vorhandenen Modellelemente kein Problem darstellt, ist die präzise Vorhersage des Rechen- und Kommunikationsaufwands meist eine extrem schwierige Aufgabe.

Das liegt daran, dass man das exakte Verhalten des Modells zu Beginn der Simulation noch nicht kennt - ansonsten wäre die Simulation an sich ja unnötig. Aus dem selben Grund kann die Dynamik eine Modells nicht ausreichend bei der Partitionierung beachtet werden. Es ist zum Beispiel durchaus möglich, dass ein Element des Modells kurz nach der Ausführung eine zufällige Anzahl von weiteren Elementen erzeugt.

Um dieses Manko auszumerzen, sollte während der Simulation das bereits in der Einleitung erwähnte *Load Balancing* stattfinden. Dabei handelt es sich im Grunde um eine wiederholte Repartitionierung des aktuell ausgeführten Modells, wobei beim Load Balancing viele neue Fragestellungen auftreten. Da diese Problematik in der Arbeit nicht weiter verfolgt wird, seien hier einige interessante Aspekte genannt, welche das Load Balancing von der Modellpartitionierung unterscheiden:

- Im Vordergrund steht natürlich der offensichtliche Vorteil gegenüber der (initialen) Partitionierung: bei einer *Re*-Partitionierung kann auf die zuvor erhaltenen Ergebnisse zurückgegriffen werden. Es muss jedoch geklärt werden, inwiefern diese Ergebnisse überhaupt wiederverwendet werden können, ohne die strukturellen Änderungen, die unterdessen im Modell vor sich gingen, zu vernachlässigen.

- Ein Load Balancing - Algorithmus hat gegenüber einem Modellpartitionierungsalgorithmus den Vorteil, dass er für die Abschätzung des zukünftigen Berechnungsaufwands von Modellelementen auf eigene Messungen zurückgreifen kann.

- Ein zusätzlicher Aspekt ist die benötigte Abschätzung des Aufwands, der für die Umverteilung einer Simulation betrieben werden muss, während der Aufwand der Verteilung bei der Modellpartitionierung keine große Rolle spielt. Man stelle sich eine vorhandene Verteilung vor, die nur suboptimal

ist. Stellt sich bei der Berechnung einer besseren Verteilung heraus, dass allein die Übertragung der zu verschiebenden Teile viel aufwendiger ist als der durch die gleichmäßigere Verteilung gewonnene Zeitgewinn, muss dies vermieden werden.

- Außerdem sollte sichergestellt sein, dass die Frequenz, mit der das Load Balancing durchgeführt wird, mit den Eigenschaften der Simulation (Anzahl der Untermodelle, Berechnungsaufwand) und denen der Umgebung (Anzahl und Art der Verbindungen zwischen den Prozessoren) korrespondiert. Ein zu oft durchgeführtes Load Balancing erhöht den Rechenaufwand und verlangsamt dadurch die Simulation. Wird der Ausgleich dagegen zu selten vorgenommen, können größere Ungleichgewichte über längere Zeiträume auftreten. Hier müssten geeignete Heuristiken untersucht werden (Z. B. die Ausführung des Load Balancing nach einer bestimmten Anzahl von Änderungen an der Modellstruktur, etc.).

Im Folgenden wird zwar gelegentlich auf die diese Aspekte des Load Balancing eingegangen eingegangen - vor allem um gewisse softwaretechnische Entscheidungen auch auf langfristige Sicht plausibel zu machen - einen entsprechenden Mechanismus zu entwickeln um die genannten Herausforderungen zu meistern würde jedoch den Rahmen dieser Arbeit bei Weitem übersteigen.

Trotz der zahlreichen und ernüchternden theoretischen Grenzen, an die man bei der näheren Beschäftigung mit dem Partitionierungsproblem stößt, gibt es auch positive Facetten.

Die vielseitige Formulierbarkeit des Grundproblems bietet die Möglichkeit, eine Ganze Reihe Ansätze aus den verschiedensten Bereichen zur Lösung einzusetzen, genannt seien hier vor allem die Mathematische Optimierung und die Theoretische Informatik.

Eine weitere Möglichkeit ist das Ausnutzen des Wissens über einzelne *Modellierungsformalismen*. Zum Beispiel bilden *Zelluläre Automaten* ein Gitternetz. Dieser Graph ist natürlich viel einfacher zu partitionieren als ein Graph von beliebiger Komplexität. Noch viel versprechender stellt sich die Situation beim *DEVS* - Formalismus dar, auf dessen Unterstützung bei der Entwicklung von JAMES II besonders viel Wert gelegt wurde.

Hier werden atomare Modellelemente zu neuen Modellelementen aggregiert. Durch diese hierarchische Struktur sind die Elemente eines DEVS - Modells baumförmig angeordnet. Ein Baum ist wiederum nur ein Graph mit sehr vielen speziellen Eigenschaften, was eine signifikante Reduzierung des Problems darstellt. Die für diese Arbeit wichtigen Eigenschaften des DEVS-Formalismus werden in Abschnitt 1.3 erläutert.

Die Ausnutzung der Besonderheiten einzelner Formalismen ist zwar insofern unbefriedigend, als dass man für jeden Modellierungsformalismus einen neuen Algorithmus entwickeln muss, der die entsprechenden Merkmale des Formalismus zur besseren Partitionierung hinzuzieht. Aber dies erscheint schon allein unter Beachtung der theoretischen Schranken des allgemeinen Partitionierungsproblems als eine durchaus lohnende Alternative.

Bevor dieser Gedanke am konkreten Beispiel einer Algorithmenentwicklung zur DEVS-Partitionierung in James II illustriert wird, sei hier noch einmal auf die große Vielseitigkeit der denkbaren Lösungsansätze aufmerksam gemacht.

Daher werden in Kapitel 2 zunächst einige Ansätze aus den verschiedensten Gebieten der Informatik und Mathematik vorgestellt. Einige davon sind allgemeiner Natur, andere - wie die Partitionierung von symmetrischen Strukturen mit Automorphismengruppen von Graphen oder die kostenbasierte Partitionierung von DEVS-Modellen - beziehen sich auf spezielle Klassen von Graphen.

1.3 Begriffsklärung: DEVS

In dieser Arbeit wird besonders der Modellierungsformalismus DEVS (Discrete EVent specified Systems) im Vordergrund stehen, da die Mehrzahl der für JAMES II erstellten Modelle auf diesem Formalismus basiert. DEVS ist ein diskret-ereignisorientierter Ansatz, der durch eine systemtheoretische Sicht auf Systeme als „Black Boxes“, die über Ein- und Ausgaben mit ihrer Umwelt interagieren, angeregt wurde.

Grundlegend ist dabei zunächst das so genannte atomare Modell: Ein atomares Modell verfügt über eine Ein- und Ausgabeschnittstelle, einen Zustand und zwei Zustandsübergangsfunktionen: Die erste, δ_{ext}, ändert den Zustand des Modells auf Grund von eintreffenden Eingaben bzw. Ereignissen, während die andere, δ_{int}, dem Modell erlaubt, seinen Zustand nach einer gewissen Zeit, die in der Simulation verstrichen ist, selbstständig zu ändern.

Um komplexere und modulare Modelle zu entwickeln, kann man atomare Modelle zu gekoppelten Modellen aggregieren. Dabei werden die atomaren Modelle, die miteinander kommunizieren sollen, über ihre Schnittstellen, genauer gesagt über so genannte *Eingabe- und Ausgabeports*, miteinander verkoppelt. Die Definition aller enthaltenen atomaren Modelle sowie aller zwischen ihnen vorhandenen Kopplungen beschreibt ein gekoppeltes Modells vollständig. Hinzu kommt, dass sich gekoppelte Modelle nach außen wie atomare Modelle verhalten, so dass diese Verschachtelung von Modellelementen beliebig fortgesetzt werden kann.

Besonders interessant beim DEVS-Formalismus ist nun, dass zusätzlich zum eigentlichen Formalismus auch die operationale Semantik des Modells, also das Verhalten bei der Ausführung, genau definiert ist. Neben vielen Details, unter anderem in [29] nachlesbar, sind für das Verständnis dieser Arbeit folgende Punkte hervorzuheben:

- Modelle kommunizieren nur mit den ihnen direkt übergeordneten Modellen, das heißt die Kommunikation erfolgt streng hierarchisch. Die Kopplungen zwischen Modellen sind nur dem direkt übergeordneten gekoppelten Modell bekannt.

  Andere Kopplungen zwischen Modellen, beispielsweise zwischen Modellen, die nicht das gleiche übergeordnete Modell besitzen, sind nicht zugelassen. Die gekoppelten Modelle sind für die den Kopplungen entsprechende Weiterleitung von Nachrichten verantwortlich.

- Aus der streng hierarchischen Kommunikation ergibt sich eine baumförmige Struktur aus Komponenten, welche die verschiedenen Teilmodelle des Modells simulieren. Dabei werden atomare Modelle mit *Simulator*-Komponenten (bzw. Simulatoren), und gekoppelte Modelle mit *Koordinator*-Komponenten (bzw. Koordinatoren) assoziiert. Simulatoren und Koordinatoren unterscheiden sich jedoch lediglich in den

Ausführungsdetails und zeigen nach außen hin identisches Verhalten. Stellt man Simulatoren und Koordinatoren als Knoten eines Graphen dar, und zeichnet man genau dann eine Kante zwischen zwei Komponenten, wenn diese direkt miteinander kommunizieren dürfen, so erhält man den so genannten *abstrakten Simulatorbaum*, der die Hierarchie aller vorhandenen Komponenten des Modells abbildet.

- DEVS ist ein diskret-ereignisorientiertes System, das heißt pro Zeitpunkt dürfen nur endlich viele Ereignisse innerhalb der Simulation auftreten. Der Zeitpunkt, zu dem ein Ereignis geschieht, kann jedoch auf einer kontinuierlichen Skala frei gewählt werden. Um die Simulation auszuführen, erzeugt der Wurzelknoten des abstrakten Simulatorbaums - auch als *Root Coordinator* bezeichnet - eine so genannte * - Nachricht. Diese Nachricht wird entlang des Simulatorbaums zu dem atomaren Modell weitergeleitet, bei dem als nächstes ein (internes, daher zeitgesteuertes) Ereignis geschieht. Bei Eintreffen der *-Nachricht führt das Modell den internen Zustandsübergang mit δ_{int} aus, und übergibt etwaige Ausgaben an das übergeordnete Modell. Von dort aus werden dann in Abhängigkeit der vorhandenen Kopplungen weitere Teile des abstrakten Simulatorbaums „aktiviert“, diese reagieren dann auf externe Ereignisse (die Eingabedaten, die über die Kopplungen propagiert werden).

  Jeder Simulationsschritt erfordert also viel Kommunikation, ausgehend von der Wurzel des Baumes. Die Kommunikation ist ausschließlich auf Komponenten, die ineineinander verschachtelt sind, beschränkt - wie bereits erwähnt ist Kommunikation nur entlang der Kanten des Simulatorbaums möglich.

Auf diese wichtigen Eigenschaften wird in verschiedenen Teilen der Arbeit wiederholt Bezug genommen. Die operationale Semantik der DEVS-Modelle ist noch weitaus komplexer als hier dargestellt. Der Einfachheit halber wurden hier nur die Aspekte des Formalismus diskutiert, die für das Verständnis der Arbeit wesentlich sind.

1.4 Terminologie

Damit die in den folgenden Abschnitten ausgeführten Überlegungen gut verstanden werden, sollen an dieser Stelle die wichtigsten verwendeten Begriffe genauer beschrieben werden.

Mit *Modellgraph* werden Graphen bezeichnet, welche Struktur und Komponenten des zu partitionierenden Modells repräsentieren. Wie diese Graphen aufgebaut sind und wie sie erstellt werden ist den nachfolgenden Kapiteln zu entnehmen. Ein *Modellbaum* ist ein Modellgraph, der die Baumeigenschaft besitzt, das heißt er hat keine Kreise und ein Wurzelknoten ist definiert. Die Existenz eines Wurzelknotens mag für die graphentheoretische Definition von Bäumen nicht erforderlich sein, in dieser Arbeit wird allerdings vorausgesetzt, dass für alle Bäume ein solcher Wurzelknoten definiert ist.

Analog zu Modellgraphen symbolisieren *Infrastrukturgraphen* die vorhandenen Prozessoren und deren Kommunikationsverbindungen.

Der Begriff *Prozessor* wird im Folgenden oftmals als Synonym für Computer genutzt. Für die in dieser Arbeit vorgestellten Verfahren ist es irrelevant, ob es

sich bei dem Prozessor um die Komponente eines Großrechners handelt, oder um einen einzelnen Computer.

Eine *Partition* ist eine Zerlegung eines Modells (bzw. des Modellgraphen oder des Modellbaums) in verschiedene, paarweise disjunkte *Partitionsblöcke*, oder einfach *Blöcke*. Die Blöcke werden im Text des Öfteren an Stelle der *Prozessoren* verwendet, welche sie symbolisieren. Blöcke werden daher gelegentlich auch mit den Eigenschaften der Prozessoren - wie Rechenkapazität etc. - assoziiert. Mit *Partitionierung* ist der eigentliche Erstellungsprozess einer Partition gemeint.

Ebenen eines Baumes sind jeweils Mengen bzw. *Tupel* von Knoten, die alle die gleiche *Pfadlänge* zum Wurzelknoten besitzen. Ist eine Ebene *höher* gelegen als eine andere Ebene, haben ihre Knoten eine kürzere Pfadlänge zur Wurzel. Bäume werden, wie in der Informatik weit verbreitet, mit dem Wurzelknoten an höchster Position dargestellt.

Beschränkungen bzw. Einschränkungen des Suchraums durch den Nutzer werden mit dem englischen Fachwort *Constraints* bezeichnet, wie in der Literatur zum Thema üblich. Dies wurde der besseren Verständlichkeit halber beibehalten.

Weitere Fachbegriffe, die für das Verständnis dieser Arbeit wichtig sind, aber keine zentrale Rolle einnehmen, werden im Glossar ab Seite 106 beschrieben.

1.5 Verwendete Formatierungen

Zur besseren Lesbarkeit werden in den nachfolgenden Kapiteln bestimmte Formatierungen der Schrift verwendet. `Schreibmaschinen-Schrift` wird für Programmkonstrukte in Algorithmen oder zur Bezeichnung von implementierten Programmteilen verwendet.

Kursive Schrift kann die Hervorhebung einer Tatsache bedeuten, wie zum Beispiel bei „Dieser Algorithmus ist *nicht* korrekt“. Sind dagegen *Fachbegriffe* kursiv dargestellt, so können diese im Glossar nachgelesen werden, oder es handelt sich dabei um neue Begriffe, die an Ort und Stelle erklärt werden und nur für den entsprechenden Abschnitt relevant sind.

Kapitel 2

Vorhandene Ansätze

2.1 Überblick

Die in diesem Abschnitt vorgestellten Ansätze zur Entwicklung praktikabler Partitionierungsmethoden sollen vor allem die Vielseitigkeit der dafür vorhandenen Ideen illustrieren. Aus diesem Grund werden auch einige etablierte Verfahren, wie beispielsweise *Simulated Annealing* [13], zugunsten weniger bekannter Vorgehensweisen außen vor gelassen. Die Auswahl der hier besprochenen Methoden darf also keineswegs als vollständig oder repräsentativ angesehen werden.

Im letzten Abschnitt werden die Methoden dann zusammengefasst und - soweit dies möglich ist - miteinander verglichen. Dies ist jedoch nicht immer einfach, da viele der vorgestellten Ansätze Annahmen über das Modell bzw. den Modellgraphen treffen, und damit für bestimmte Klassen von Modellen oder Modellstrukturen besonders geeignet sind.

Generell lassen sich alle Ansätze zur Partitionierung anhand zweier Eigenschaften kategorisieren, die in jeweils zwei Ausprägungen vorliegen können. Die erste Eigenschaft bezieht sich auf die Art der Suche nach einer Partition: Eine *globale Suche* sucht im gesamten Suchraum nach der besten Partition, während eine *lokale Suche* nur in der Umgebung einer Ausgangspartition nach besseren Partitionen sucht. Die Ausgangspartition kann dabei im vorigen Iterationsschritt berechnet worden sein, oder zuvor mit einem anderen Verfahren erzeugt werden.

Die zweite Eigenschaft bezieht sich auf die Verarbeitung der Graphen selbst. Dabei wird zwischen Ansätzen, die auf dem eingegebenen Graphen nach einer guten Partition suchen, und den so genannten *Multilevel* - Algorithmen unterschieden. *Multilevel* - Algorithmen versuchen, das Problem in mehrere leichtere Teilprobleme zu zerlegen, indem sie den Graphen zunächst vereinfachen.

Bestimmte Teilgraphen werden dabei als einzelne Knoten aufgefasst, bis eine hinreichend einfache Struktur vorliegt. Dies kann mehrmals wiederholt werden, so dass man verschiedene „Abstraktionsebenen“ des Graphen erhält, auf denen dann, ebenenweise herabsteigend, nur noch die jeweils in der vorigen Ebene abstrahierten Elemente einem Partitionsblock zugeordnet werden müssen.

Diese Idee ist vor allem dann erfolgreich, wenn man die Teilgraphen des Graphen zu einzelnen Knoten abstrahiert, bei denen die Wahrscheinlichkeit besonders groß ist, dass sie in einer guten Partition des Graphen alle dem gleichen Partitionsblock zugeordnet wären. Cliquen von Graphen, also Mengen von Kno-

ten, die alle miteinander benachbart sind, sind ein gutes Beispiel für diese Art Teilgraph.

Da die Grundproblematik jedoch bestehen bleibt, also auch ein *Multilevel* - Algorithmus anfangs einen - wenn auch einfacheren - Graphen partitionieren muss, wird im Folgenden nur auf Algorithmen eingegangen, die auf dem eingegebenen Graphen arbeiten.

Faszinierend an der Graphenpartitionierung sind vor allem die viele Berührungspunkte zu anderen Forschungsgebieten. So gibt es neben klassischen Ansätzen aus der Optimierung, wie der Nutzung von *Genetischen Algorithmen* (GA), auch Ansätze aus der algebraischen Graphentheorie oder der Physik.

In Abbildung 2.1 ist dies veranschaulicht. Hinsichtlich der Abbildung muss noch festgestellt werden, dass im Grunde genommen alle Verfahren für die Modellpartitionierung verwendet werden, und damit im Zusammenhang zur Modellierung und Simulation stehen. Der in Abschnitt 2.5 beschriebene Ansatz ist in der Abbildung direkt mit diesem Gebiet verbunden, weil er außerdem die Eigenschaften spezieller Modellierungsformalismen bei der Partitionierung berücksichtigt, und daher diesbezüglich aus dem Feld der vorgestellten Ansätze hervorsticht.

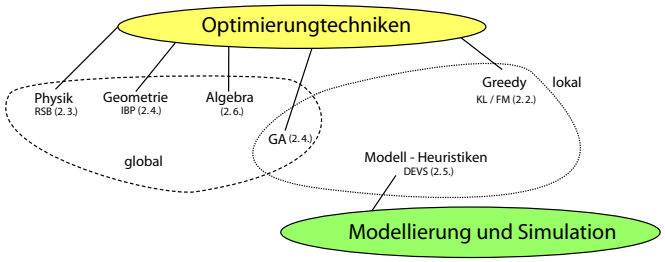

Abbildung 2.1: Einordnung der hier vorgestellten Partitionierungsverfahren in globale und lokale Methoden sowie angrenzende Forschungsgebiete

Die Tatsache, dass es derart viele Gebiete gibt, die Ansätze zur Graphenpartitionierung hervorgebracht haben, liegt auch darin begründet, dass man die Partitionierung von Graphen nicht nur zur Partitionierung von Modellen, sondern zum Beispiel auch zum verteilten Rechnen mit *sparse* - Matrizen nutzen kann. Dabei handelt es sich um Matrizen, die wenige Elemente besitzen, welche ungleich Null sind.

Sparse Matrizen sind oftmals sehr groß und treten zum Beispiel bei physikalischen Berechnungen recht häufig auf. Ihre effiziente verteilte Berechnung gelingt jedoch nur, wenn man die Elemente der Matrix, mit denen ein Prozessor rechnen soll, so geschickt verteilen kann, dass möglichst wenig Kommunikation zwischen den Prozessoren daraus resultiert.

Dies ist der Fall, wenn möglichst viele der Elemente, die ungleich Null sind, in der Nähe der Hauptdiagonale der Matrix angeordnet sind. Das ist leicht einzusehen, denn verteilt man beispielsweise die Matrizenmultiplikation auf verschiedene Prozessoren, können Matrizen dieser Art die Kommunikation zwischen den Prozessoren stark reduzieren: Weil fast alle Werte, die zur Berechnung von Belang sind, in der Nähe der Hauptdiagonale liegen, müssen nur wenige weitere Informationen über andere Werte in den Spalten oder Zeilen eingeholt werden.

Die Anordnung der Elemente nahe der Hauptdiagonale einer Matrix kann durch geschicktes Umsortieren der Spalten und Zeilen erreicht werden, der Fachausdruck hierfür ist die *Verringerung der Bandbreite* einer Matrix.

Interpretiert man eine *sparse* - Matrix nun als die Adjazenzmatrix eines Graphen, so würde eine optimale Partitionierung des Graphen dieses Problem lösen, und umgekehrt.

2.2 Das KL - Verfahren

Dieses in seiner Ursprungsform von Kernighan und Lin entwickelte lokale Verfahren zur Partitionierung wird meist als KL-Algorithmus bezeichnet [1] und ist eines der klassischen Verfahren zur Graphenpartitionierung.

Es nimmt eine rekursive Bisektionierung des Graphen vor. Eine Bisektion ist eine Partitionierung des vorliegenden Graphen in genau zwei Partitionsblöcke. Der Algorithmus selbst erzeugt aus einer gegebenen Bisektion, der *Startbisektion*, eine neue Bisektion, die eine kleinere Schnittgröße besitzt. Die Eingabebisektion kann dabei zufällig gewählt worden sein, oder wird von einem anderen Verfahren erzeugt. Soll der Graph in mehr als zwei Blöcke aufgeteilt werden, kann jeder der einzelnen Teile erneut diesem Verfahren unterzogen werden, es lässt sich also rekursiv auf den erhaltenen Ergebnissen fortsetzen.

Der Algorithmus optimiert dabei nur die Schnittgröße, also die Kommunikationskosten zwischen den zwei Blöcken. Unterschiede bei den Knotengewichten werden nicht beachtet - stattdessen wird die Anzahl der Knoten pro Partition berücksichtigt.

Diese Art der rekursiven Bisektionierung ist ein gutes Beispiel für einen Algorithmus, der die Partition lediglich durch lokale Änderungen verbessert. Sie wurde von vielen Seiten aufgegriffen und erweitert, wie noch genauer ausgeführt wird.

Das Grundprinzip des Algorithmus ist eigentlich recht einfach: Sei $\{N_1, N_2\}$ die Startbisektion des Graphen $G = (V, E)$. Es gilt also $N_1 \cap N_2 = \emptyset, N_1 \cup N_2 = V$. Die Funktion $w(v, u) : V \times V \rightarrow \mathbb{R}$ gibt das Kantengewicht der Kante zwischen u und v an. $P(v) : V \rightarrow \mathbb{N}$ gibt die Nummer des Partitionsblocks an, dem der Knoten v zugewiesen wurde.

Es wird nun für jeden Knoten berechnet, wie sich die Schnittgröße zwischen beiden Partitionsblöcken verändert, wenn dieser vom einen Block in den anderen verschoben werden würde. Für jeden Knoten $v \in V$ aus $G = (V, E)$ ist

$$int(v) = \sum_{(v,u) \in E \wedge P(v) = P(u)} w(v, u)$$

die Funktion zur Berechnung des Ausmaßes der Kommunikation mit Knoten im selben Block. Analog wird mit

$$ext(v) = \sum_{(v,u) \in E \wedge P(v) \neq P(u)} w(v, u)$$

die Kommunikation des Knotens zu den Knoten außerhalb seines Partitionsblocks ermittelt.

Nun wird für alle Knotenpaare (v_1, v_2) mit $v_1 \in N_1, v_2 \in N_2$ errechnet, wie viel man an Schnittgröße zwischen den Partitionsblöcken einspart, wenn man den einen Knoten mit dem anderen vertauscht.

Die Veränderung der Schnittgröße beim Verschieben eines einzelnen Knotens $v \in V$ ergibt sich aus

$$g(v) = ext(v) - int(v).$$

Für zwei Knoten $v_1 \in N_1, v_2 \in N_2$ berechnet man nun den *gain*, also den *Zugewinn* an Partitionsqualität bezüglich der Schnittgröße bei Vertauschung dieser beider Knoten mit

$$gain(v_1, v_2) = \begin{cases} g(v_1) + g(v_2) - 2w(v_1, v_2), wenn (v_1, v_2) \in E \\ g(v_1) + g(v_2) \end{cases}.$$

Hierbei ist zu beachten, dass das doppelte Kantengewicht einer gemeinsamen Kante von der Summe der Knoten-*gains* abgezogen werden muss. Dies ist damit zu erklären, dass in $g(v)$ alle Kantengewichte von Kanten in die andere Partition *positiv* bewertet werden (und zwar bei beiden Knoten), obwohl sich eine gemeinsame Kante im Fall einer Vertauschung der Knoten noch immer im Schnitt zwischen den Blöcken befände. Deshalb darf die Kante zwischen den beiden untersuchten Knoten keine Rolle bei der Abwägung einer guten Vertauschung spielen, und ihr Einfluss wird durch den zusätzlichen Term nichtig gemacht.

Im Folgenden werden nur diese Vertauschungen zweier Knoten betrachtet, wodurch gesichert ist, dass die Größe der Partitionsblöcke konstant bleibt. Es wird also eine Menge von Knotenpaaren gesucht, deren Vertauschung eine Verringerung der Schnittgröße mit sich bringt.

Dazu werden die Zugewinne bei Vertauschung aller möglicher Knotenpaare berechnet und in einer Zugewinn-Matrix gespeichert. Dann wählt man $n = \min(|N_1|, |N_2|)$ -mal den jeweils größten Zugewinn und tauscht die beiden zugehörigen Knoten aus, speichert die neue Bisektion und aktualisiert die Zugewinn-Matrix. Bereits vertauschte Knoten werden markiert, so dass sie nicht mehr vertauscht werden können. Am Ende erhält man eine n Einträge lange Liste mit Knotenvertauschungen $(v_1^i, v_2^i), i = 1, \ldots, n$, und deren Zugewinn. Jetzt sucht man ein j, so dass

$$\sum_{i=1}^{j} gain(v_1^i, v_2^i)$$

maximal ist. Ist diese Summe positiv, so werden die Vertauschungen der Knotenpaare (v_1^i, v_2^i) für $i = 1, \ldots, j$ an der Startbisektion durchgeführt, und der Algorithmus beginnt, mit der neu entstandenen Bisektion als Startbisektion, von vorn. Ansonsten terminiert der Algorithmus.

Die aufwendigste Operation bei diesem Verfahren ist offensichtlich die Berechnung der Zugewinn-Matrix und deren Aktualisierung nach jedem Schritt. Dies kann jedoch durch das Ausnutzen der Tatsache beschleunigt werden, dass nur die Werte der Nachbarknoten des zu vertauschenden Knotenpaares aktualisiert werden müssen. Bereits markierte Knoten können ebenfalls ausgelassen werden. Das j, für das der Zugewinn bei den Vertauschungen maximal ist, lässt sich durch eine einfache lineare Suche in der Liste sehr schnell finden. Insgesamt ergibt sich eine Zeitkomplexität von $O(|V|^3)$ für jeden Iterationsschritt, die allerdings noch verringert werden kann [1].

Wichtig ist, dass der Algorithmus *immer* die Vertauschung von bis zu n Knotenpaaren abwägt, auch wenn der Zugewinn zwischenzeitlich negativ ist, sich also die Qualität der Bisektion durch die Vertauschung verschlechtern würde. Dies geschieht dann in der Hoffnung, in einer der nächsten Vertauschungen umso mehr davon zu profitieren. Der Algorithmus versucht dadurch, das „Hängenbleiben“ an lokalen Maxima zu vermeiden.

Im Grunde genommen handelt es sich also um ein *Greedy*-Verfahren, dass mit einer Suchtiefe von n Vertauschungen die größtmögliche Verbesserung der Partitionsqualität sucht. Falls sich diese dann als positiv herausstellt (Zugewinn-Summe der Vertauschungen ist > 0), wird vom neu erreichten Zustand aus mit dem nächsten Iterationsschritt begonnen.

Bildlich gesprochen ist dieses Verfahren eine Form der Optimierung mittels „Bergsteigen im Nebel“: Vom Ausgangspunkt aus wird ein gewisser Teil des Suchraums untersucht. Findet sich eine höher gelegene Stelle innerhalb der untersuchten Umgebung, begibt man sich dorthin und beginnt von Neuem.

Zwar liefert der Algorithmus meist annehmbare bis gute Ergebnisse [6], aber leider ist seine Zeitkomplexität von $O(|V|^3)$ ein starker Nachteil bei größeren Graphen. Er hat daher in seiner ursprünglichen Form kaum noch praktische Relevanz.

Allerdings wurde das Grundprinzip durch viele verschiedene Erweiterungen verbessert - zum Beispiel durch das Verschieben lediglich eines Knotens pro Iterationsschritt, wodurch die Zeitkomplexität pro Iteration auf $O(|E|)$ verringert werden konnte [4]. Der in Frage kommende Block für den Knoten wird dabei immer abwechselnd gewählt, so dass die Größendifferenz beider Blöcke maximal 1 beträgt. In [5] wird ein daran angelehnter Algorithmus beschrieben, der außerdem noch Knotengewichte berücksichtigt, sowie eine beliebige Menge an Partitionsblöcken. Weitere Modifikationen des Originalalgorithmus sind in [6] evaluiert worden.

2.3 Rekursive spektrale Bisektion

Die rekursive spektrale Bisektion (RSB) ist ein Verfahren, das sowohl über die lineare Algebra als auch mittels physikalischer Betrachtungen, das heißt Differentialgleichungssysteme, motiviert werden kann. Beide Herleitungen sollen hier grob vorgestellt werden, die Beweise zu den Sachverhalten würden in diesem Rahmen jedoch zu weit führen.

Auf der beiliegenden CD befinden sich Materialien, in denen weiter gehende Erklärungen und Herleitungen skizziert werden. Der eigentliche algebraische Ansatz wurde von Pothen, Simon und Liu ([7], [8]) entwickelt.

Zunächst wird jedoch die physikalische Idee hinter diesem Verfahren vorgestellt. Danach wird gezeigt, wie sich die Lösung für das entstandene Differentialgleichungssystem mit Mitteln der linearen Algebra bestimmen lässt.

Grundidee zur Entwicklung der RSB ist die physikalische Berechnung einer gespannten Saite, die mit einer bestimmten Frequenz schwingt. Wenn eine Saite mit der Frequenz $f = 2$ schwingt, beschreibt sie eine Sinus-Kurve. Die eine Hälfte der Saite wird bezüglich ihrer Ruheposition nach oben, und die andere dementsprechend nach unten ausgelenkt.

Diese „natürliche“ Einteilung wird nun auch für den Graphen $G = (V, E)$ gesucht. Die Knoten des Graphen werden dabei als Gewichte angesehen, die alle

gleich schwer sind. Statt Kanten sind die Gewichte mit Federn verbunden. Nun wird versucht, die Auslenkung der einzelnen Gewichte zu berechnen, wenn das Gebilde mit der Frequenz $f = 2$ schwingen soll (siehe Abbildung 2.2). Diese Aufteilung würde dann eine Partition des Graphen liefern.

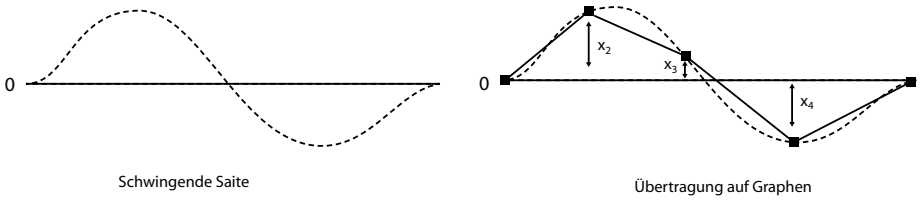

Abbildung 2.2: Skizze zur Grundidee der rekursiven spektralen Bisektion: Links ist eine schwingende Saite angedeutet, rechts davon der Beispielgraph, auf den dieses Prinzip übertragen wird.

Es wird zunächst von einem Weg der Länge n als Graphen ausgegangen. Wie in Abbildung 2.2 zu sehen ist, besitzt dieser Graph große Ähnlichkeit mit einer Saite. Später kann das Verfahren dann für beliebige Graphen verallgemeinert werden.

Die Endpunkte des Graphen, also $v_1, v_n \in V, |V| = n$, werden als fixiert angenommen. Nun kann man, ausgehend von der Newtonschen Mechanik, ein Differentialgleichungssystem entwickeln, das als Variablen die Auslenkungen der verschiedenen Knoten besitzt:

Sei D die so genannte *Federkonstante* der Federn zwischen den Gewichten, und seien $x_1, \ldots, x_n \in \mathbb{R}$ die Auslenkungen aus der Horizontale für jeden Knoten $v_1, \ldots, v_n \in V$ (siehe $x_2, \ldots, x_4$ in Abbildung 2.2, $x_1 = x_5 = 0$). Es wird weiterhin angenommen, dass die Feder von v_i nach v_{i+1} proportional zur Differenz $x_{i+1} - x_i$ gestreckt wird. Ist diese Differenz positiv, so wirkt die Federspannkraft $F_S = D \cdot s$, wobei s die Dehnung der Feder bezeichnet, nach oben, ansonsten nach unten. Die Gewichtskraft, die auf einen Knoten v_i wirkt, lässt sich für kleine Auslenkungen mit

$$F^G_{v_i} = D \cdot (x_{i-1} - x_i) + D \cdot (x_{i+1} - x_i) = D \cdot (x_{i-1} - 2x_i + x_{i+1})$$

approximieren, das heißt es werden die Federspannkräfte der Federn zu seinen benachbarten Knoten addiert. Setzt man diese Approximation ins Newtonsche Grundgesetz $\vec{F} = m \cdot \vec{a}$ ein, erhält man unter Berücksichtigung von $\vec{a} = \frac{\delta \vec{v}}{\delta t} = \frac{\delta^2 \vec{s}}{\delta t^2}$ und $x_1 = x_n = 0$:

$$m \cdot \frac{\delta^2 \bar{x_i}(t)}{\delta t^2} = F^G_{\bar{v_i}} = -D \cdot \begin{pmatrix} 2x_1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ \cdots \\ -x_{n-1} + 2x_n \end{pmatrix} =$$

$$-D \cdot \begin{pmatrix} 2 & -1 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 & 0 \\ & \vdots & & \vdots & & \vdots & \\ 0 & \cdots & 0 & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} =$$

$$-D \cdot M \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \texttt{ daraus folgt: } -\frac{m}{D} \cdot \frac{\delta^2 \bar{x_i}(t)}{\delta t^2} = M \cdot \bar{x_i}(t)$$

$\bar{x_i}(t)$ ist dabei ein Spaltenvektor, der die Auslenkung der Knoten v_i zu einem Zeitpunkt t symbolisiert. Jetzt kann man versuchen, für dieses Differentialgleichungssystem einen Lösungsvektor der Form $\bar{x_i}(t) = \sin(\bar{a} \cdot t) \cdot \bar{x_0}$ zu finden, so dass die Frequenz der Schwingung gleich 2 ist.

Diese Herleitung ist aber nicht nur auf derart einfach strukturierte Graphen anwendbar. Beispielsweise könnte man sich *planare Graphen* als eine Art „Trampolin“ vorstellen - die äußeren Knoten in der Ebene werden fixiert, die Inneren werden in Schwingung gebracht. Analog dazu lassen sich beliebige andere Graphen verwenden.

Die letzte Gleichung oben, $-\frac{m}{D} \cdot \frac{\delta^2 \bar{x_i}(t)}{\delta t^2} = M \cdot \bar{x_i}(t)$, besitzt die Form eines linearen Gleichungssystems und kann als ein *Eigenwertproblem* betrachtet werden.

Dabei handelt es sich um ein Problem, bei dem $M \cdot \chi = \lambda \cdot \chi$ für eine Matrix M gegeben ist, und die so genannten *Eigenwerte* λ_i mit den dazugehörigen *Eigenvektoren* $\chi_i (= \bar{x_i}(t))$ der Matrix M bestimmt werden müssen. Die Eigenvektoren spannen dabei den Lösungsraum des linearen Gleichungssystems auf.

Modifiziert man zusätzlich das zu Grunde liegende Gleichungssystem in Bezug auf die Gewichtskraftberechnung für Anfangs- und Endknoten, erhält man die so genannte *Laplace-Matrix* des Graphen $G = (V, E)$, $L(G)$, die sich leicht aus der Struktur des Graphen konstruieren lässt:

$$L(G)(i,j) = \begin{cases} i = j : \textit{ Grad des Knotens } v_i \in V \\ \{v_i, v_j\} \in E : -1 \\ 0 \textit{ sonst} \end{cases}$$

Dies ist für den Beispielgraphen aus Abbildung 2.2 offensichtlich, gilt aber für beliebige Graphen. Nun lässt sich sowohl physikalisch als auch algebraisch herleiten, dass die gesuchten Werte für eine Zweiteilung gleich dem Eigenvektor χ_2 zum zweitkleinsten Eigenwert λ_2 von $L(G)$ sind. Damit kann die RSB effizient berechnet werden. Analog kann man die Eigenvektoren der nächstgrößeren Eigenwerte für die Aufteilung in $p = 3, \ldots n$ Partitionen verwenden.

Die eigentliche Abarbeitung des Algorithmus ist also sehr einfach: Zunächst wird $L(G)$ bestimmt, dann wird mit geeigneten Verfahren der zweitkleinste Eigenwert λ_2 und der dazugehörige Eigenvektor χ_2 bestimmt. Nun wird jeder Knoten $v_i \in V$ einem Partitionsblock zugewiesen: Ist χ_2 an Stelle i positiv, wird der Knoten i dem einen, ansonsten dem anderen Block zugewiesen.

Ähnliche Algorithmen, die auch die physikalischen Gesetzmäßigkeiten zur Lösung von Problemen der Informatik nutzen, findet man zum Beispiel für das Zeichnen von Graphen [9].

Dieses Verfahren ist dafür bekannt, Partitionen von sehr hoher Qualität zu liefern. Leider ist die Berechnung der benötigten Komponenten sehr aufwendig und das Verfahren damit für größere Graphen eher ungeeignet. Ironischerweise benötigt man zur effizienten verteilten Berechnung des Algorithmus leistungsfähige Matrizenoperationen.

Dies setzt wiederum die Bandbreitenreduzierung der Matrizen voraus, was, wie in Abschnitt 2.1 ausgeführt wurde, dem Partitionierungsproblem selbst entspricht.

2.4 Genetische Algorithmen

Da die Partitionierung von Graphen auch als reines Optimierungsproblem betrachtet werden kann (siehe Abschnitt 1.2), wurden viele interessante Heuristiken der linearen Optimierung darauf angewendet, unter Anderem auch genetische Algorithmen. Im hier beschriebenen Ansatz von Maini et al. [10] wird aufgezeigt, dass diese Vorgehensweise sogar zum Load Balancing genutzt werden kann.

Genetische Algorithmen (GA) stellen eine Untermenge der *Evolutionären Algorithmen* dar. Dies sind Algorithmen, die gewisse Elemente der Evolutionstheorie aufgreifen um bestimmte Probleme zu lösen. Sie wurden ursprünglich von Holland vorgeschlagen [11].

Die Probleme, auf die genetische Algorithmen angewendet werden, zeichnen sich meist durch einen extrem großen Suchraum aus. Dies ist beispielsweise der Fall, wenn zu wenig über die Charakteristika einer optimalen Lösung bekannt ist.

Generell wird bei einem genetischen Algorithmus versucht, aus einer vorhandenen, beispielsweise zufällig gewählten, Grundmenge von Lösungsvorschlägen mit Hilfe von in der Evolutionstheorie postulierten Verfahren neue und bessere Lösungen zu finden. Durch diese Zielsetzung kann ein genetischer Algorithmus auch als Optimierungsverfahren betrachtet werden.

Dabei wird jeder Lösungsvorschlag als ein Individuum angesehen, dass eine gewisse *Fitness*, also Tauglichkeit bezüglich des zu lösenden Problems, besitzt. Dieses Maß der Tauglichkeit kann als eine Funktion aufgefasst werden, deren globales Optimum bestimmt werden soll. In der Terminologie der linearen Optimierung ist die Fitness-Funktion also die Zielfunktion des Optimierungsproblems. Die Menge aller Individuen wird als (Gesamt-)Population bezeichnet, die anfangs vorhanden Individuen bilden die Startpopulation.

Die Lösungsansätze selbst, also die Individuen, sind lediglich Datenstrukturen, die konkrete Belegungen für die Eingabeparameter der Fitnessfunktion liefern. Der Inhalt dieser Datenstrukturen - von der ja die Fitness des jeweiligen Individuums abhängig ist - wird folgerichtig auch als *Genom* des Individuums bezeichnet.

Mit genetischen Algorithmen versucht man nun, aus den vorhandenen Lösungsvorschlägen neue Vorschläge zu generieren. Die Qualität neuer Vorschläge kann mit Hilfe der Fitnessfunktion eingeschätzt werden. Wird eine Lösung von zufriedenstellender Qualität gefunden, bricht der Algorithmus ab.

Das Besondere an genetischen Algorithmen ist die Art und Weise, wie aus alten Lösungsvorschlägen neue generiert werden. In der Evolutionstheorie werden

hauptsächlich drei Einflussfaktoren für das Entstehen immer besser angepasster Lebewesen genannt:

- *Selektion*: Selektion bedeutet, dass die Individuen mit der größten Fitness die größten Überlebenschancen haben. Somit haben sie auch die besten Chancen, ihr Erbgut in das Genom eines Individuums der darauf folgenden Generation einzubringen (*„The fittest will survive“*).

- *Rekombination*: Bei der geschlechtlichen Fortpflanzung geht nicht das komplette Genom eines Elternteils auf das Kind über, sondern die Genome beider Elternteile werden zu einem neuen Genom kombiniert. Dieser auch als *Crossover* bezeichnete Prozess kombiniert die Eigenschaften der beiden Elterngenome miteinander.

- *Mutation*: Die Mutation ist ein „Störfaktor“ bei der Übertragung eines Genoms auf ein neu entstehendes Individuum. Anders als die Rekombination, bei der lediglich vorhandene Genome auf neue Weise miteinander kombiniert werden, wird durch die Mutation ein Teil des Genoms vom neuen Individuum gänzlich verändert. Durch Mutation wird also ein in gewissen Regionen „neues“ Genom erzeugt, wodurch das Entstehen sehr verschiedener Lösungsansätze ermöglicht wird. Aus Sicht der Optimierung handelt es sich also um eine Erweiterung des Suchraums in eine zufällig gewählte Richtung, während in der Biologie von der Erhöhung der genetischen Diversität gesprochen wird.

Eine Adaption dieser Vorgehensweisen wird von einem genetischen Algorithmus dazu genutzt, eine Evolution möglicher Lösungen innerhalb einer mathematisch definierten Umgebung zu simulieren. Die Qualität eines genetischen Algorithmus wird daher entscheidend durch die Umsetzung dieser drei fundamentalen Konzepte bestimmt. Außerdem spielen noch Anzahl und Qualität der Individuen in der Startpopulation, sowie die Anzahl der Generationen, über die eine Evolution der Lösungsvorschläge simuliert wird, eine wichtige Rolle.

Bei der Selektion gibt es einige Verfahren, die sich im Detail unterscheiden. Generell wird den Individuen mit höherer Fitness eine höhere Chance eingeräumt, sich zu vermehren. Meist dient die Selektion also lediglich der Auswahl von Elternindividuen zur Erstellung der nächsten Generation von Lösungsvorschlägen. Dabei werden die Eltern zufällig gewählt, wobei die Wahrscheinlichkeit, dass ein bestimmtes Individuum „gezogen“ wird, von seiner Fitness abhängt.

Bei der Rekombination ist es besonders kritisch, wie oft und an welchen Stellen die vorhandenen Genome miteinander kombiniert werden. Wird zu stark rekombiniert, dann werden gerade die Charakteristika, welche die Fitness der als Eltern gewählten Individuen gesteigert haben, eventuell wieder zerstört - *obwohl* sich die beiden Individuen fortpflanzen konnten. Wird die Rekombination dagegen zu zu wenig genutzt, werden nicht genug verschiedene Kombinationen erstellt und große Teile des Suchraums werden nicht untersucht.

Auch die Festlegung der *Mutationsrate*, also der Wahrscheinlichkeit, dass sich ein einzelnes Gen eines neuen Individuums verändert, ist heikel: Wird die Mutationsrate zu hoch angesetzt, entwickelt der Algorithmus eher das Verhalten einer zufälligen Suche im Suchraum, da auch hier die Fitness der Eltern kaum Auswirkungen auf die Fitness der Kinder haben wird. Eine zu kleine Mutationsrate

führt aber zu einer starken Einschränkung des Suchraums, denn dadurch wird das Entstehen von bisher unbekannten Lösungsvorschlägen unwahrscheinlicher.

Trotz dieser Probleme bieten genetische Algorithmen den großen Vorteil, relativ effizient einen großen Suchraum zu durchsuchen. Deshalb bietet sich dieses Verfahren auch für die Suche nach einer guten Partition an.

In [10] gehen Maini et al. noch einen Schritt weiter, indem sie versuchen, dass Wissen über das vorhandene Problem in die Evolutionsstrategie zu integrieren. Ihr Verfahren soll hier am Beispiel einer *Bisektion* erklärt werden, ist aber prinzipiell zur Partitionierung in beliebig viele Blöcke nutzbar.

Durch die Beschränkung auf zwei Blöcke zur Partitionierung des Graphen $G = (V, E)$ wird vor allem die Notation der Genome stark vereinfacht. Geht man von einer geordneten Liste der Knotenmenge V aus, $|V| = n$, kann man Bitfolgen der Länge n, also Wörter aus $\{0, 1\}^n$, betrachten.

Dabei entscheidet die Belegung des i-ten Bits b_i, ob sich der Knoten i im ersten Partitionsblock ($b_i = 0$) oder im Zweiten ($b_i = 1$) befindet. „001“ beschreibt also als Lösung die Bisektion eines Graphen mit 3 Knoten, so dass sich die Knoten 1 und 2 im Ersten, und der Knoten 3 im zweiten Partitionsblock befinden.

Als Selektionsverfahren wurde eine *Ranking - Selektion* genutzt. Diese nutzt nicht die eigentliche Fitness der vorhandenen Individuen aus, sondern erstellt anhand der Fitness eine Rangliste, in der dann die vorderen Plätze bevorzugt werden. Anschaulicher formuliert vermeidet dieses Verfahren das Aussterben krasser Außenseiter und das zu starke Beeinflussen der nächsten Generation durch ein einzelnes, sehr leistungsstarkes Individuum. Die Ordnung zwischen den Individuen bleibt zwar bestehen, aber es spielt keine Rolle, ob das beste Individuum 1.5 mal oder 10.000 mal leistungsfähiger als das zweitbeste Individuum ist. Die Mutationsrate wurde auf 1% gesetzt.

Interessant ist auch Umsetzung der Rekombination. Es gibt sehr verschiedene Arten von Rekombinationsverfahren. Das einfachste, ursprünglich von Holland vorgeschlagene Verfahren legt lediglich *einen* zufällig gewählten Rekombinationspunkt fest, so dass jedes Elternteil in zwei Teile zerlegt werden kann. Diese Teile werden untereinander ausgetauscht, so dass dadurch zwei neue Individuen entstehen: $A.B + C.D \Rightarrow A.D, C.B$.

Dies ist natürlich ungenügend, wenn die Genome eine gewisse Größe erreichen, im hier beschriebenen Fall also bei der Partitionierung von größeren Graphen. Dieses Konzept lässt sich aber verallgemeinern, indem man die Anzahl der Rekombinationspunkte proportional zur Länge der Genome wählt.

Mit dem *Uniform Crossover* geht man nun noch einen Schritt weiter, indem die Anzahl der Rekombinationspunkte variabel gehalten wird. Für jedes Element des Genoms, hier also für jedes Bit, wird zufällig entschieden, von welchem Elternteil es die entsprechende Belegung erben soll.

Man kann sich die Rekombination in diesem Fall als die Generierung und Anwendung eines Bitmusters vorstellen: Zunächst wird eine zufällige Bitfolge $b = (b_1, \ldots, b_n), |V| = n$ generiert.

Die Wahrscheinlichkeit, dass das ein Bit gleich Null bzw. gleich Eins ist, ist *gleichmäßig* verteilt, das heißt sie liegt für jedes Bit bei jeweils 50% - daher der Name dieses Operators. Anhand dieser Bitfolge wird dann entschieden, welches Bit von welchem Elternteil übernommen wird. Erzeugt man die Bitfolge „001101“, ergibt sich daraus eine Rekombination mit 3 Rekombinationspunkten (nach der 2., 4. und 5. Stelle), und das Kind-Individuum erhält die ersten beiden

Bits sowie das fünfte vom ersten Elternteil, und die restlichen Bits vom Zweiten.

Mit Hilfe dieses Crossover-Ansatzes kann man nun versuchen, Wissen über die Struktur des Graphen in den Algorithmus zu integrieren. Dies gelingt, indem man die beim *Uniform Crossover* noch gleichmäßig verteilte Wahrscheinlichkeit des Erbens beeinflusst. Dazu kann man die Position des zufällig gewählten Bits sowie Informationen über die Elterngenome ausnutzen.

Das daraus resultierende Verfahren wird auch als ***k**nowledge-based **n**on-**u**niformal **cross**over* (**KNUX**) bezeichnet. Kann das Crossover-Verfahren im Verlauf des Algorithmus sogar noch weiter verfeinert werden, spricht man von einem ***d**ynamischen* KNUX (**DKNUX**).

Doch welches Wissen könnte in so einen Rekombinationsmechansismus integriert werden? Im Allgemeinen muss versucht werden, die Qualität der Elterngenome positionsweise zu schätzen, und jeweils dem an dieser Stelle vorteilhaften Genom eine höhere Wahrscheinlichkeit zuzuweisen. Doch wie kann man im Voraus abschätzen, welches Erbgut sich schließlich als das Beste herausstellt?

Die Antwort auf diese Frage mag anfangs absurd klingen, wird aber näher erläutert: Angenommen, man hätte bereits eine von einem weiteren Algorithmus, beispielsweise dem KL-Verfahren, erzeugte Lösung von relativ guter Qualität, also eine Art „Ausgangspartition“. Dann könnte die Rekombinationsoperation diese Lösung bei der Erzeugung der Rekombinations-Bitfolge b hinzuziehen.

Um dies zu erreichen, wird eine so genannte *Bias - Matrix* P (bias: engl., Tendenz) angelegt. Sie besitzt $n = |V|$ Zeilen und je eine Spalte für jeden Partitionsblock. An Position (i, j) wird der Anteil der Nachbarknoten von $v_i \in V$ gespeichert, die bei der qualitativen Lösung dem Block j zugewiesen wurden. Die Elemente dieser Matrix werden, wie weiter unten zu sehen ist, als Wahrscheinlichkeiten zur Auswahl des vererbenden Elternteils genutzt.

Ist der Knoten i in einem Elternteil dem Block k zugeordnet, so ist die Wahrscheinlichkeit, dass das Kind an Stelle i von diesem Elternteil erbt, gleich $P(i, k)$. Die Wahrscheinlichkeiten der beiden Elternindividuen müssen bei mehr als 2 Blöcken allerdings noch interpoliert werden, so dass ihre Summe 1 ergibt.

Zusammenfassend gilt also zur Berechnung des Kindgenoms vor der Mutation:

```
a = (a_1, ..., a_n)  // Genom des ersten Elternteils
b = (b_1, ..., b_n)  // Genom des zweiten Elternteils
c = (c_1, ..., c_n)  // Genom des Kindes
P[1...n][1...p]  // Bias - Matrix für p Partitionsblöcke

for (i=1; i < n; i++) {

    //Wenn gleiche Information, Auswahl irrelevant
    if (a_i == b_i)
        p_i = 1
    else
        p_i = P_[i,a_i] / (P_[i,a_i] + P_[i,b_i])  //Interpolation

    if(rand() < p_i)
        c_i = a_i
    else
```

$c_i = b_i$

}

Danach wird noch der Mutationsoperator auf dem Genom des Kindindividuums ausgeführt. Hat man auf diese Weise genügend Kindindividuen erzeugt, ist eine neue Generation an Lösungsvorschlägen vorhanden. Diese bilden diese die neue Gesamtpopulation, gegebenen Falls zusammen mit einigen besonders guten Elternindividuen. Auf dieser neuen Population kann dann mit der Erzeugung einer weiteren Generation begonnen werden.

Da sich die *Bias - Matrix* an einer vor Ausführung des Algorithmus ermittelten Partition orientiert, die höchstwahrscheinlich nicht optimal war, sollte sie in gewissen Abständen neu berechnet werden. Statt der am Anfang gewählten Partition wird dazu der Lösungsvorschlag des bis dahin besten Individuums verwendet, falls dessen Lösung besser ist.

So kann auch zur Laufzeit gewonnenes Wissen um mögliche Strukturen einer optimalen Lösung beim Crossover berücksichtigt werden. Auf diese Weise kann ein für die Graphenpartitionierung optimierter **DKNUX** definiert werden, und es gelingt Maini et al., die Optimierung mit genetischen Algorithmen an die Problemstellung der Graphenpartitionierung anzupassen.

Als großer Nachteil dieser Methode muss jedoch gelten, dass zuvor ein weiterer Algorithmus zur Erzeugung einer guten Ausgangspartition benötigt wird. Somit ist diese Methode eher als Verbesserungsmethode für bereits vorhandene Partitionsvorschläge zu verstehen. In den Tests des Algorithmus wird außerdem deutlich, wie abhängig er von einer guten Ausgangspartition zur Kalibrierung des Crossover-Operators ist, was die Verwendung von zufällig erzeugten Partitionen für diese Zwecke verhindert. Dadurch läuft dieser Ansatz Gefahr, sich selbst gewissermaßen ad absurdum zu führen, denn für eine gute Ausgangspartition braucht man eben einen guten Partitionierungsalgorithmus.

Zur Untersuchung des Einflusses der Ausgangspartition auf das Ergebnis wurden von Maini et al. drei Arten von Partitionen verwendet: die Ergebnisse einer zufälligen Partitionierung, die Ergebnisse einer rekursiven spektralen Bisektion (RSB, siehe Abschnitt 2.3) sowie Resultate des so genannten *Index Based Partitioning* (IBP).

Index Based Partitioning ist ein sehr einfaches Verfahren zur Graphenpartitionierung, bei dem in drei Schritten vorgegangen wird: Zunächst wird allen Knoten ein Index zugewiesen, so dass die (geometrische) Lage der Knoten untereinander weitestgehend erhalten bleibt. Das Finden einer geeigneten Heuristik für diesen Schritt ist die größte Herausforderung bei dieser Methode. Danach werden alle Knoten ihrem Index nach sortiert und die sortierte Liste wird in p Partitionsblöcke zerlegt. Dieser Algorithmus ist also nur anwendbar, wenn „geometrische“ Informationen über die Knoten des Graphen überhaupt vorliegen. Es liefert jedoch akzeptable Ergebnisse und ist viel schneller als die RSB. In Abschnitt 4.1 wird eine spezielle Form des IBP für die entwickelten Partitionierungskomponenten vorgestellt.

Zum Testen des genetischen Algorithmus werden damit ein zufälliges Ergebnis, ein mittelmäßiges (IBP) und ein sehr gutes (RSB) als Ausgangspartition angenommen. Damit kann man zeigen, dass dies einen starken Einfluss auf die Qualität des Endergebnisses hat: So ist unter Verwendung einer RSB-Partition eine Reduzierung der Kanten zwischen den Partitionsblöcken um bis zu 10 % im Vergleich zur Ausgangspartition möglich. Nutzt man die IBP - Partitionierung

zur Initialisierung, erreicht der Algorithmus nur in der Häflte der Fälle ein besseres Ergebnis als die RSB-Startpartition, und meist ist die Verbesserung sehr klein. Bei einer zufällig gewählten Ausgangspartition kann das Ergebnis sogar schlechter sein als das Ergebnis einer RSB.

Dennoch gibt es einen interessanten Vorteil dieses Algorithmus gegenüber anderen Ansätzen: Er arbeitet sehr gut, wenn es um die Neuberechnung einer vorhandenen Partition aufgrund von Veränderungen im Graphen geht. Das lässt sich damit erklären, dass die günstigsten Strukturen im Restgraphen bereits vorher gefunden wurden, und mit genetischen Algorithmen optimal wiederverwendet werden können. In den Tests wurden ca. 10 - 20% neue Knoten an einer zufällig gewählten Stelle hinzugefügt, und trotzdem konvergierte der Algorithmus sehr viel schneller als bei der initialen Partitionierung. Dies könnte das Verfahren sehr nützlich für Systeme machen, die sich dynamisch verändern und damit Load Balancing - Mechanismen benötigen.

Leider sind diese Testergebnisse für endgültige Aussagen nicht ausreichend, da nur das Hinzufügen von Knoten zu einem Graphen getestet wurde, nicht aber das Herauslöschen etc..

Es sei noch bemerkt, dass Maini et al. auch eine parallele Implementation des Algorithmus vorstellen - die allerdings kaum Vorteile bringt - sowie ein *Hill-climbing* für Individuen nach jedem Evolutionsschritt, wodurch lokale Optima besser erreicht werden.

Alles in allem erscheint dieser Ansatz zwar viel versprechend, aber recht aufwendig. Er weist leicht verbesserte Ergebnisse auf, ist allgemeingültig und steuert interessante Erkenntnisse zur Integration von domänenspezifischem Wissen in genetische Algorithmen bei. Wirklich praxistauglich scheint er jedoch nicht zu sein, dafür ist er wohl zu aufwändig, und die Qualität der Ergebnisse variiert stark in Abhängigkeit der Anzahl der Generationen, der Anzahl der benötigten Partitionen, der Ausgangspartition und der Größe des Graphen.

Eine andere Anwendungsmöglichkeit von evolutionären Algorithmen zur Graphenpartitionierung stellt die Partitionierung mittels simulierter Ameisenkolonien dar, wobei das Verhalten der Ameisen mit Hilfe genetischer Programmierung ermittelt wurde [12].

2.5 DEVS - Partitionierung mit Kostenfunktionen

Das in diesem Abschnitt beschriebene Verfahren von Zeigler et al. [14] ist die Einzige der hier vorgestellten Methoden, welche gewisse Besonderheiten eines Modellierungsformalismus zur Entwicklung eines effizienteren Algorithmus ausnutzt. Gleichzeitig verwendet die Methode auch Kostenfunktionen, weshalb durchaus Parallelen zur linearen Optimierung und ihren Zielfunktionen bestehen.

Durch die Verallgemeinerung einiger Aspekte auf ein Kostenmaß ist es möglich, diesen Algorithmus prinzipiell auf viele andere Modellierungsformalismen anzuwenden, an dieser Stelle wird er trotzdem unter dem Aspekt der DEVS-Modellpartitionierung behandelt, da die Partitionierung von DEVS-Modellen für JAMES II besonders wichtig ist, und Zeigler et al. diese als Beispiele für das Verfahren explizit verwenden.

Voraussetzung zur Nutzung des Algorithmus bleibt jedoch, dass das zu partitionierende Modell hierarchisch aufgebaut ist. Der Algorithmus arbeitet daher nicht auf beliebigen Graphen, sondern ausschließlich auf Bäumen.

Kernpunkt des Algorithmus ist zunächst die Schätzung der Kosten für die einzelnen Modellelemente. Dies ist natürlich abhängig vom verwendeten Modellierungsformalismus, daher wird hier nur auf DEVS-Modelle eingegangen: Es werden sowohl atomare als auch gekoppelte Modelle mit Kosteninformationen versehen. Gekoppelte Modelle speichern zusätzlich die Gesamtkosten ihrer Submodelle, damit der Algorithmus später nicht den gesamten Modellbaum untersuchen muss, wenn er Partitionen zuweist.

In [14] werden drei Kostenmaße für DEVS vorgestellt, die verschiedene Kriterien mit einbeziehen:

- *Ein- und Ausgänge:* Es wird davon ausgegangen, dass die Anzahl der Ein- und Ausgabeports eines Modells im Allgemeinen proportional zum Berechnungsaufwand des Modells ist.
- *Komplexität des Modells:* Es wird angenommen, dass die Anzahl der möglichen Zustände eines Modells im Allgemeinen proportional zum seinem Berechnungsaufwand ist.
- *Systemaktivität:* Bei gewissen Modellen könnte man auch davon ausgehen, dass die Anzahl der internen Zustandsänderungen (engl.: Transitions) innerhalb eines gewissen Simulationszeitraums ausschlaggebend für ihren Berechnungsaufwand ist. Diese Anzahl kann man ermitteln, indem man die Modelle über einen festgelegten Zeitraum simuliert, keine Eingaben erzeugt und jeweils die Anzahl der internen Zustandsübergänge durch δ_{int} zählt.

Zwar hat keine dieser Behauptungen Anspruch auf Allgemeingültigkeit, aber es ist durchaus möglich, dass jedes der drei Maße zumindest bei bestimmten Modellen gilt. Dies müsste dann allerdings vom Modellierer vor der Ausführung spezifiziert werden. Die Kostenmaße könnten ansonsten auch, wie im Beispiel weiter unten gezeigt, multiplikativ verknüpft werden um eine Kostenfunktion zu erhalten, in die alle Maße eingehen.

Von der Verschiedenheit der Modelle und der daraus resultierenden Unsicherheit beim Gebrauch von Kostenmaßen abgesehen, ist selbst die Ermittlung der dafür benötigten Daten nicht immer einfach: Beispielsweise lassen sich in *JAMES II* Ein- und Ausgabeports sowie die Systemaktivität durchaus messen, aber da es sich bei δ_{int} und δ_{ext} um programmierte Methoden handelt, die lediglich bestimmte Zustandsvariablen ändern, lässt sich die theoretische Anzahl der möglichen Zustände nicht ohne Decompilierung und enormen Analyseaufwand ermitteln. Für *JAMES II* sähe daher eine mögliche Kostenfunktion für ein Modell m folgendermaßen aus:

$$cost(m) = |I_m| \cdot |O_m| \cdot Transitions(m),$$

wobei I_m bzw. O_m die Menge der Ein- bzw. Ausgabeports bezeichnet, und $Transisitions(m)$ die Anzahl der internen Zustandsübergänge für einen gewissen Zeitraum angibt.

Die genaue Analyse zur Güte der vorgeschlagenen Maße würde hier zu weit führen, wie in Abschnitt 3.3 näher ausgeführt wird.

Es sei aber noch angemerkt, dass das Abzählen interner Zustandsübergänge eigentlich eine besondere Art von Pre-Simulation zur Kostenabschätzung darstellt, genaueres dazu ist im Überblick über weitere Ansätze auf Seite 31 nachzulesen.

Wenn im Folgenden Kosten von Knoten Erwähnung finden, ist damit der über eine zu definierende Kostenfunktion geschätzte Rechenaufwand für das Modell gemeint, das der Knoten repräsentiert. Ist von Kosten bezüglich Partitionsblöcken die Rede, ist dies die Summe der Kosten aller im Block enthaltenen Knoten bzw. Modelle.

Der Algorithmus unterscheidet zwei Phasen: Zum einen eine Phase zur Erzeugung einer Ausgangspartition, und später dann die schrittweise Verbesserung dieser Ausgangspartition.

Um die Ausgangspartition zu erstellen, wird der *Modellbaum* des DEVS-Modells, also die Repräsentation des Modells als Graph, von der Wurzel aus betrachtet. Dazu erstellt man eine Liste aller Kindknoten der Wurzel. Dann wird die Länge dieser Liste mit der Anzahl der zu erstellenden Partitionsblöcke, p, verglichen. Ist die Anzahl der Knoten in der Liste kleiner als p, wird ein Knoten *expandiert*, das heißt er wird aus der Liste gelöscht, und seine Kindknoten werden stattdessen in die Liste aufgenommen. Zur Expansion wird immer ein Knoten gewählt, der die höchstmöglichen Berechnungskosten aufweist. Eine weitere Bedingung an einen expandierbaren Knoten ist natürlich, dass er Kindknoten besitzt, er muss also ein gekoppeltes Modell im Modellbaum repräsentieren.

Sobald mehr als p Knoten in der Liste sind, werden die p teuersten Knoten zu jeweils einem der p Blöcke hinzugefügt.

Die restlichen Knoten der Liste werden so aufgeteilt, dass der jeweils am wenigsten kostende Knoten dem Partitionsblock mit den geringsten Kosten zugewiesen wird.

Damit liegt bereits eine Ausgangspartition vor. Die nicht direkt vom Algorithmus betrachteten Knoten werden dem Block zugewiesen, in dem ihr Vorfahr liegt.

Leider geben die Artikel von Zeigler et al. keinen Aufschluss über die Zuweisung der expandierten Knoten, sie könnten aber zum Beispiel dem Block mit den niedrigsten Kosten zugewiesen werden.

In der zweiten Phase des Algorithmus, dem so genannten *Evaluation-Expansion-Selection(E^2S) - Partitioning*, wird innerhalb eines Iterationsschrittes eine neue Partition erstellt. Deren Güte wird dann mit einer Evaluierungsfunktion berechnet, was wieder als Analogie zur mathematischen Optimierung angesehen werden kann. Ähnlich wie beim KL-Verfahren wird auch hier abgebrochen, wenn der vollzogene Iterationsschritt keine Verbesserungen mit sich bringt. Es besteht also auch hier die Gefahr, an gewissen lokalen Optima hängen zu bleiben.

Im Iterationsschritt selbst wird zunächst von der zuletzt berechneten Partition ausgegangen. Am Anfang wird die in der ersten Phase erzeugte Ausgangspartition verwendet.

Nun wird der Partitionsblock mit den höchsten Kosten gesucht, der einen expandierbaren Knoten enthält. Gibt es in dieser Partition mehrere solcher Knoten, wird derjenige mit den höchsten Kosten gewählt. Dann wird der Knoten wie bei der Erstellung einer Ausgangspartition expandiert.

Dies bedeutet, er wird zunächst aus dem Block gelöscht. Hatte der Block

nur diesen einen Knoten, so verbleibt der teuerste Kindknoten des gelöschten Knoten im Block. Alle anderen Kindknoten werden analog zum Verfahren zur Erzeugung der Ausgangspartition auf die vorhandenen Blöcke aufgeteilt: Der kostengünstigste Knoten wird dem Block zugewiesen, dessen Kosten die geringsten sind.

Wie bereits erwähnt wird diese neu entstandene Partition nun von einer Evaluierungsfunktion geprüft. Wenn sie sich als Verbesserung herausstellt, wird sie als Ausgangspartition für den nächsten Iterationsschritt verwendet. Andernfalls terminiert der Algorithmus. Durch dieses strenge Abbruchskriterium ist es möglich, dass der Algorithmus auf ein lokales Minimum stößt und zu früh abbricht.

Als Evaluierungsfunktion nutzen Zeigler et al. die Kostendifferenz zwischen dem kostenintensivsten und -günstigsten Partitionsblock.

In [15] wird der Algorithmus noch verfeinert, vor allem indem der Vorgang des Expandierens mit Hilfe des Parameters k an die praktischen Gegebenheiten angepasst werden kann. Der Parameter k gibt an, wie viele Ebenen des Modellbaums bei der Expansion eines Knotens expandiert werden sollen. Im ursprünglichen Algorithmus ist $k = 1$, das heißt die direkten Kinder des gewählten Knotens werden auf die anderen Blöcke verteilt. Ist $k = 2$, werden statt dessen die Kinder aller Kinder des gewählten Knotens verteilt, usw. .

Der Ansatz von Zeigler et al. hat viele Vorteile: Durch die Beschränkung des Algorithmus auf Bäume können auch sehr große Modelle extrem schnell partitioniert werden, da der Algorithmus nur so lange Knoten expandiert, bis keine weitere Verbesserung der Partition mehr erreicht werden kann. Die in [15] vorgestellten Ergebnisse zeigen weiterhin, dass die Partitionen bezüglich der Evaluierungsfunktion von extrem hoher Qualität sind, insbesondere für $k = 2$ und $k = 3$.

Der einzige Nachteil des Algorithmus ist, dass die Evaluierungsfunktion leider *nur* die Kostendifferenz zwischen den Partitionsblöcken misst. Dies ist ein ziemlich großer Nachteil, weil die Geschwindigkeit einer DEVS-Simulation auch stark von der Kommunikationsgeschwindigkeit *zwischen* den Modellen abhängen kann. Daraus folgt, dass Eltern- und Kindknoten nach Möglichkeit im gleichen Block liegen sollten. Auf diesen Sachverhalt nimmt der Algorithmus keine Rücksicht.

2.6 Ausnutzung von Symmetrien

Der letzte Ansatz, der im Rahmen dieser Arbeit vorgestellt werden soll, fällt auf Grund seiner Exotik etwas aus dem Rahmen. Allerdings kann man an diesem Beispiel sehr gut erkennen, wie unterschiedlich die Herangehensweisen an dieses Problem sein können.

Bei diesem von Lemeire et al. [16] vorgestellten Ansatz werden Erkenntnisse der Graphen- und Gruppentheorie herangezogen, um Symmetrien im gegebenen Modell zu erkennen und diese zur Partitionierung zu nutzen. Diese Vorgehensweise birgt den großen Vorteil in sich, dass die Zahl der sinnvoll zu erstellenden Partitionen hier automatisch erkannt werden kann, wobei eine „natürliche“ Einteilung des Modellgraphen vorgenommen wird.

Wie die Überschrift schon vermuten lässt, ist dieses Verfahren auf relativ symmetrische Modelle ausgerichtet. Dies ist aber keine starke Einschränkung,

da sehr viele Modelle Symmetrien aufweisen. Besonders gut lassen sich damit Modelle aus dem Bereich des Hardwareentwurfs partitionieren, worauf dieses Verfahren auch ausgerichtet ist.

Doch zunächst zu den mathematische Begriffen, die im Folgenden verwendet werden. Gegeben sei ein Graph $G = (V, E)$. Ein *Automorphismus* f ist eine Abbildung $f : V \to V$, mit

$$\forall v_1, v_2 \in V : \{v_1, v_2\} \in E \Leftrightarrow \{f(v_1), f(v_2\} \in E$$

Umgangssprachlich bildet ein Automorphismus eine Selbstähnlichkeit - oder Symmetrie - des Graphen ab, indem er die Knoten eines Graphen auf andere Knoten desselben Graphen abbildet, ohne dass sich ihre Eigenschaften, ausgedrückt durch die Kanten zwischen ihnen, dabei ändern. Ein Knoten ist nach der Anwendung eines Automorphismus auf den Graphen immer noch mit den gleichen Knoten benachbart, mit denen er vor der Anwendung benachbart war. Eine Veranschaulichung dieses Sachverhalts ist in Abbildung 2.3 dargestellt.

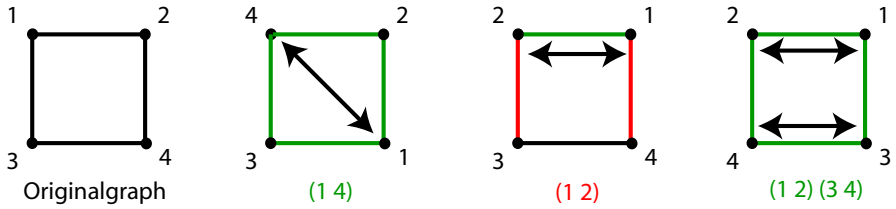

Abbildung 2.3: Beispiele für Automorphismen von Graphen: Links ist der Originalgraph abgebildet. Die Pfeile zwischen Knoten sollen die darunter definierten Abbildungen andeuten.

Tauscht man die Knoten 1 und 4 des Originalgraphen in Abbildung 2.3 aus, so sind beide immer noch jeweils mit den Knoten 2 und 3 benachbart. Damit hat der Graph die gleichen Eigenschaften wie der Originalgraph.

Wie in der Skizze zu sehen, gibt es für diese Art von Abbildung zwischen Knoten eine eigene, als *zyklisch* bekannte Schreibweise. Danach bedeutet „(14)“, dass Knoten 1 auf Knoten 4, und dieser wiederum auf Knoten 1 abgebildet wird. Die restlichen Knoten werden auf sich selbst abgebildet.

Diese Notation kann man nun hintereinander fügen, damit bedeutet „(12)(34)“, dass jeweils die Knoten 1 und 2 sowie 3 und 4 vertauscht werden. Gleichzeitig kann dies auch auf mehrere Knoten ausgedehnt werden, das heißt „(abc)“ würde für eine Abbildung stehen, die a auf b, b auf c und c schließlich wieder auf a abbildet.

Da ein Automorphismus die Eigenschaften jedes Knotens erhält, kann man zwei vorhandene Automorphismen auch hintereinander ausführen und erhält wieder einen Graphen mit den Eigenschaften des Originalgraphs.

Durch die Ausführung zweier Automorphismen hintereinander wird implizit ein neuer Automorphismus erzeugt. Das Hintereinanderausführen kann daher als binärer Operator auf der Menge aller Automorphismen eines Graphen interpretiert werden.

Jeder Graph besitzt mindestens einen trivialen Automorphismus, der jeden Knoten auf sich selbst abbildet. Die Menge aller Automorphismen eines Gra-

phen bildet zusammen mit der Operation der Hintereinanderausführung die so genannte *Automorphismengruppe* des Graphen, als $Aut(G)$ bezeichnet. Die Umkehrung jeder Abbildung stellt dabei ihr inverses Element dar, und das Hintereinanderausführen ist assoziativ und kommutativ. Das Einselement der Gruppe ist der triviale Automorphismus. Näheres über die verwendeten Begriffe aus der Gruppentheorie kann in [26] nachgelesen werden.

Der *Orbit* eines Knotens $v \in V$ ist folgendermaßen definiert:

$$orb(v) = \{g(v) | g \in Aut(G)\}.$$

Anders ausgedrückt ist der Orbit eines Knotens die Menge an Knoten, auf die ein einzelner Knoten durch einen Automorphismus abgebildet werden kann. Da zu jedem Automorphismus ein (bezüglich Hintereinanderausführen) inverser Automorphismus existiert, liegt ein Knoten in den Orbits aller Knoten, die in seinem eigenen Orbit liegen. Dadurch ergibt sich eine Zerlegung des Graphen, wie sie in Abbildung 2.4 an einem Beispielgraphen dargestellt ist. Orbits werden im Folgenden auch als Synonym für eine durch diese Zerlegung definierte Teilmenge von Knoten des Graphen verwendet.

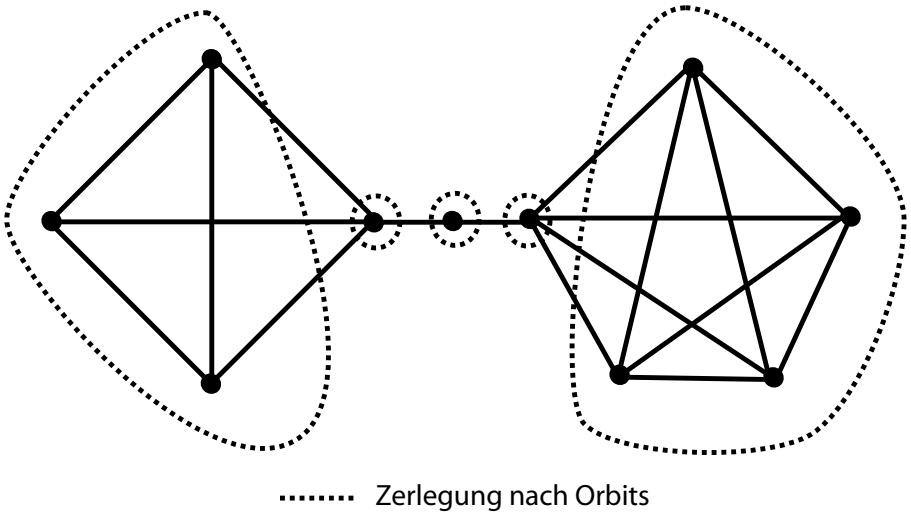

Abbildung 2.4: Beispiele für die Zerlegung eines Graphen nach Orbits: Man kann erkennen, dass die Zerlegung nach Orbits den Graph in Mengen untereinander austauschbarer Knoten zerlegt.

Zum Verständnis des Algorithmus müssen noch zwei weitere Begriffe definiert werden: Analog zum Begriff in der Gruppentheorie sei $Aut(G)_{V'}$ der *Stabilisator* einer Teilmenge $V' \subseteq V$. Dies ist die Menge der Automorphismen, die alle Knoten aus V' auf sich selbst abbilden, also unverändert lassen:

$$Aut(G)_{V'} = \{g \in Aut(G) | \forall v \in V' : g(v) = v\}.$$

Weitaus wichtiger ist die Definition eines so genannten *Block of Imprimitivity*, eine Teilmenge $B \subseteq V$ von Knoten, für die

$$\forall g, g' \in Aut(G) : g(B) = g'(B) \vee g(B) \cap g'(B) = \emptyset$$

gilt. Es handelt sich dabei um Mengen von Knoten, die für jeden Automorphismus des Graphen komplett in sich selbst symmetrisch sind ($g(B) = g'(B)$) oder auf einen gänzlich anderen Teil des Graphen abgebildet werden ($g(B) \cap g'(B) = \emptyset$). Keine Teile des Graphen, auf die ein Block of Imprimitivity abgebildet werden kann, dürfen einander überlappen.

Darauf aufbauend ist vor allem die Definition eines *Maximal Block of Imprimitivity* von Bedeutung. Dies ist eine maximale Teilmenge $B \subset V$, welche die Eigenschaften eines Block of Imprimitivity aufweist. Im Folgenden werden nur noch Maximal Blocks of Imprimitivity betrachtet, die innerhalb eines Orbits V' liegen. Im Trivialfall $|V'| = 1$ gilt $B = V'$.

Die Kernidee besteht darin, diejenigen Maximal Blocks of Imprimitivity, die in benachbarten Orbits liegen, zu einer symmetrischen Einheit - einem *Symmetrieblock* - zusammenzufassen. Orbits gelten als benachbart, wenn eine Kante zwischen ihnen existiert.

Diese Symmetrieblöcke dürfen allerdings nur dann gebildet werden, wenn keiner der maximalen Blöcke Kanten zu weiteren im jeweils anderen Orbit befindlichen maximalen Blöcken besitzt. Außerdem darf keiner der Blöcke bereits einem anderen Symmetrieblock zugeordnet sein. Dies wird durch Abbildung 2.5 am Beispielgraphen veranschaulicht, und weiter unten genauer erklärt.

Wie ist dieses recht abstrakte Vorgehen nun zu motivieren? Neben der anschaulicheren Erklärung wird hier eine grobe Beweisskizze aufgezeigt, da diese für das tiefere Verständnis des Ansatzes dienlich ist.

Die Zerlegung in Orbits gliedert den ursprünglichen Graphen in Mengen von Knoten, in denen es ähnliche Strukturen gibt. Wenn also zwei Maximal Blocks of Imprimitivity in verschiedenen Orbits liegen, dann bedeutet dies zunächst, dass kein Knoten aus dem einen Block jemals in den anderen abgebildet werden kann - sie sind sich also *unähnlich*. Das Verfahren beruht nun darauf, diese unähnlichen Strukturen so weit wie möglich miteinander zu verbinden, da man leicht zeigen kann, dass jeder Maximal Block of Imprimitivity symmetrisch zu einem anderen Maximal Block of Imprimitivity im selben Orbit ist. Dies gilt natürlich nicht im Trivialfall, bei dem ein Orbit nur einen Knoten enthält, und es folglich nur einen trivialen Maximal Block of Imprimitivity $B = V'$ gibt.

Danach können die aggregierten Symmetrieblöcke, die ja einander unähnliche Komponenten eines Graphen zusammenfassen, auf verschiedene Prozessoren verteilt werden, wobei alle nicht-trivialen Symmetrieblöcke (aggregiert aus mindestens einem nicht-trivialen Maximal Block of Imprimitivity) symmetrisch zu jeweils mindestens einem weiteren Symmetrieblock sind. Die Symmetrien werden also gefunden, und die größten symmetrischen Komponenten des Graphen werden auf verschiedene Prozessoren verteilt.

Diese Argumentation scheint zunächst sehr fragwürdig zu sein, kann jedoch recht einfach nachvollzogen werden, wenn man herleitet, dass es für jeden Maximal Block of Imprimitivity mindestens ein Pendant innerhalb eines nicht-trivialen Orbits geben muss:

Sei B ein Maximal Block of Imprimitivity im Orbit V'. In andere Teile des Graphen können Elemente aus B laut Definition eines Orbits nicht abgebildet werden, und gleichzeitig ist leicht einzusehen, dass B mindestens einmal in einen

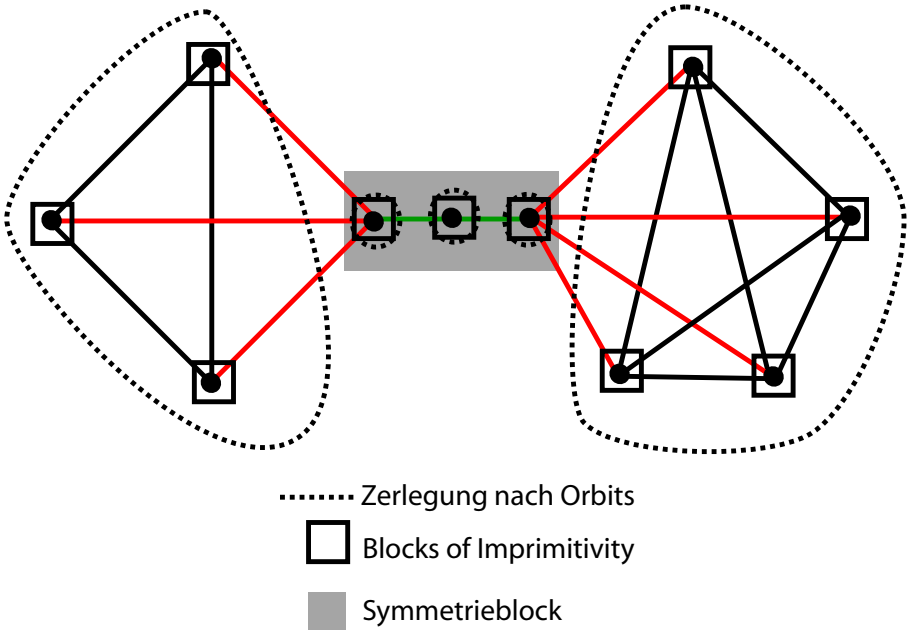

Abbildung 2.5: Nur die Knoten in der Mitte können zu einem Symmetrieblock zusammengefasst werden. Da mehrere Kanten vom rechten bzw. linken Orbit zu den mittleren Knoten existieren (rot markiert), können diese Knoten nicht zum Symmetrieblock in der Mitte hinzugefügt werden. Die eingezeichneten Blocks of Imprimitivity sind bereits alle maximal, da die Orbits entweder nur einelementig sind, oder komplette Teilgraphen enthalten - ein zweielementiger Block könnte in einem der Teilgraphen sofort auf sich überlappende Teile des Graphen abgebildet werden, was nicht erlaubt ist.

anderen Teil von V' abgebildet werden *muss* (Voraussetzung $|V'| > 1$): Zunächst muss $B \subset V'$ gelten (nach Definition eines Maximal Block of Imprimitivity). Da $B \subset V'$, gibt es mindestens einen Knoten $v' \in V'$, so dass $v' \notin B$. Allerdings liegt v' im Orbit aller Knoten von B, so dass für ein $v \in B$ ein Automorphismus g existiert mit $g(v) = v'$.

Weil aber nach Definition eines Block of Imprimitivity alle Knoten innerhalb des Blocks *entweder* auf dieselbe Knotenmenge - was ja für v nicht der Fall ist - oder aber auf eine völlig andere Knotenmenge im Graphen abgebildet werden müssen, muss es einen weiteren, zu B symmetrischen, Maximal Block of Imprimitivity innerhalb von V' geben.

Deshalb nutzt man die Maximal Blocks of Imprimitivity der Orbits, um Symmetrieblöcke zu aggregieren. Man zerlegt den Graph durch die Symmetrieblöcke in maximale Teilgraphen, die entweder keinerlei Symmetrien mit dem restlichen Graphen aufweisen, oder noch mindestens einen zu ihnen symmetrischen Teilgraphen besitzen.

Je nach gewünschter Menge an Partitionsblöcken kann jetzt jedes Orbit V', unter Nutzung des dadurch definierten Stabilisators $Aut(G)_{V-V'}$ als Automor-

phismengruppe, rekursiv dem selben Verfahren unterzogen werden, und wird dadurch in die nächstkleineren, wiederum untereinander symmetrischen oder völlig asymmetrischen Teilgraphen, aufgeteilt. So bildet sich ein so genannter *symmetry tree*, ein Symmetriebaum.

Der Symmetriebaum kann jetzt dazu genutzt werden, den Graphen auf eine beliebige Anzahl von Prozessoren zu verteilen: Sind nur wenige Prozessoren vorhanden, werden die größten symmetrischen Elemente, zum Beispiel die erste Ebene des Symmetriebaums, zur Festlegung der Partition verwendet. Bei mehreren Prozessoren kann der Baum dann ebenenweise herabgestiegen werden, das heißt an Stelle der größten symmetrischen Komponenten werden nun deren symmetrische Subkomponenten etc. auf die Prozessoren verteilt.

Leider sind noch einige Detailfragen über das Verhalten bei eher asymmetrischen Graphen ungeklärt. Außerdem wurde der Algorithmus nur auf extrem symmetrischen Varianten von Graphen getestet, was sicherlich seine Leistungsfähigkeit bei der Schaltkreissimulation demonstriert - nicht jedoch seine Anwendbarkeit auf nur „ziemlich“ symmetrische Graphen.

Herauszustellen ist aber, dass es sich hierbei um einen sehr eleganten und für bestimmte Probleme sicherlich sehr guten Algorithmus handelt, der auf einer völlig anderen mathematischen Herangehensweise beruht als die vorigen Ansätze. Außerdem ist dieses Verfahren prädestiniert für eine bestimmte Klasse von Graphen, ähnlich dem Algorithmus von Zeigler et al. in Abschnitt 2.5, der nur auf Bäumen arbeitet.

Die nächste Frage wäre nun, ob Orbits und Maximal Blocks of Imprimitivity eines Graphen halbwegs effizient ermittelt werden können. Da das Finden von Automorphismen selbst ein bekanntes Problem darstellt, existieren dafür schon einige effiziente Algorithmen, die bereits evaluiert wurden [17].

Lemeire et al. [16] verwenden den so genannten *nauty* - Algorithmus [31]. Ihre Methode wurde auf dem zu simulierenden Modell eines Netzwerk-Switches getestet. Bei den hochsymmetrischen Strukturen von Hardware kann der Algorithmus seine Stärken voll ausspielen, und dementsprechend hochwertig sind die erzeugten Partitionen.

Trotz der Verwendung optimierter Methoden für die verschiedenen Teilaufgaben benötigte der Algorithmus, auf einem 1.8 GHz Pentium 4 getestet, teilweise sehr viel Zeit - bei einem aus 4000 Knoten bestehenden Modell des erwähnten Switches, einem Graphen mit einer Vielzahl von Symmetrien, benötigte er zum Beispiel 1331 s, also ungefähr 20 Minuten. Die Komplexität hängt dabei weitestgehend vom Erkennungsalgorithmus für Automorphismen ab, ein Umstieg auf vorhandene bessere Algorithmen zur Lösung dieser Teilprobleme würde den Vorgang also zumindest etwas beschleunigen [17].

2.7 Angrenzende Forschungsgebiete

Bei der Auswahl der in den letzten Abschnitten vorgestellten Verfahren wurde besonders viel Wert darauf gelegt, das breite Spektrum möglicher Herangehensweisen zu illustrieren. Nichtsdestotrotz konnte die Menge der Forschungsgebiete, die eine große Rolle bei der Lösung des Partitionierungsproblems spielen, in diesem Rahmen nur angedeutet werden.

Neben Ideen aus der linearen Optimierung, der linearen Algebra und natürlich der theoretischen Informatik, welche den eigentlichen Ansätzen zur

Lösung des Problems zu Grunde liegen, müssen in der Praxis auch Methoden für verschiedene Teilaufgaben, die vor und nach der eigentlichen Partitionierung benötigt werden, entwickelt werden.

Dazu gehören zum Beispiel das Zeichnen von Graphen, um dem Nutzer die Ergebnisse der Partitionierung vermitteln zu können, oder die Verwendung spezialisierter Datenstrukturen für Graphen, um die Zeit- oder Speicherkomplexität der implementierten Algorithmen zu minimieren.

Ein weiteres wichtiges Thema ist das effiziente Ermitteln von Eingabedaten für den Partitionierungsalgorithmus. Obwohl die Lösung dieses Problems in dieser Arbeit nicht weiter verfolgt wird (siehe Abschnitt 3.3), soll an dieser Stelle die bereits in Abschnitt 2.5 erwähnte Methode der Pre-Simulation kurz erläutert werden.

Unter Pre-Simulation versteht man die Durchführung einer „Vor-Simulation“ über bis zu 10% der für das eigentliche Experiment angesetzten Zeitdauer. Der Hintergedanke ist dabei, dass das Verhalten von Modellelementen innerhalb dieser Zeit Aufschluß über die *generellen* Eigenschaften einzelner Elemente geben könnte.

Während Zeigler et al. in Abschnitt 2.5 vorschlagen, jedes Modellelement einzeln zu pre-simulieren, wird im Allgemeinen die gesamte Simulation über einen längeren Zeitraum ausgeführt, damit die erhobenen Messdaten eher den Tatsachen entsprechen.

Die dahinter stehende Heuristik bleibt trotzdem sehr unzuverlässig und ist nur für eine bestimmte Klasse von eher statischen Modellen verwendbar. Die Simulation von *Multiagentensystemen* - eine Anwendung, für die JAMES II unter Anderem konzipiert wurde - ist ein Beispiel für eine Klasse meist hochdynamischer und extrem schwer vorhersagbarer Modelle. Hier wäre ein Ansatz über Pre-Simulation fehl am Platz, weshalb diese Technik, die ebenfalls zur initialen Modellpartitionierung eingesetzt werden könnte, aus der Diskussion vorhandener Ansätze ausgespart wurde. Die Beschreibung eines Partitionierungsalgorithmus, der von Pre-Simulation Gebrauch macht, ist in [30] zu finden.

2.8 Zusammenfassung

Durch die Konzentration auf verschiedene Aspekte der Partitionierung ist eine abschließende Bewertung der einzelnen Verfahren sehr schwierig. Als gesichert kann gelten, dass das KL-Verfahren nicht immer die besten Partitionen liefert, und dass die rekursive spektrale Bisektion meist Partitionen von hoher Qualität ermittelt. Ein ausführlicher Vergleich von etablierten Verfahren zur Graphenpartitionierung kann in [6] nachgelesen werden.

Gut zu sehen ist jedoch, dass man mit bestimmten Einschränkungen - sei es auf symmetrische Strukturen oder auf DEVS-Modelle - erstaunlich einfache und leistungsfähige Verfahren entwickeln kann. Diese Beobachtung hat den Ausschlag für viele der in den Kapiteln 3 und 4 beschriebenen Entscheidungen gegeben.

Davon abgesehen kann dieses Kapitel mit der Einsicht beschieden werden, dass es zwar viele Ansätze zur Graphenpartitionierung gibt, aber keinen allgemeingültig guten. Entweder sind die Ergebnisse nur mittelmäßig, wie beim in Abschnitt 2.4 kurz vorgestellten Index-Based Partitioning. Oder aber die Ergebnisses sind gut, jedoch viel zu aufwendig zu berechnen.

Alles in allem veranschaulicht die Untersuchung der vorhandenen Sätze auf eindrucksvolle Weise das bekannte *No Free Lunch* - Theorem, nach dem alle Suchalgorithmen - betrachtet man die Menge *aller* möglichen mathematischen Probleme - gleich gut oder schlecht sind. Die einzige wirkungsvolle Möglichkeit, bessere Suchalgorithmen zu kreieren, bestünde damit in der Einschränkung des Problems, so weit dies möglich ist.

Kapitel 3

Vorüberlegungen zur Umsetzung

3.1 Allgemeines zu den Vorüberlegungen

Wie in Kapitel 2 zu sehen war, gibt es zwar viele Ansätze zur Modellpartitionierung, aber das Ausnutzen von zusätzlichem Wissen über den zu Grunde liegenden Formalismus scheint die einzige Möglichkeit zu sein, die generelle Leistungsfähigkeit des Algorithmus maßgeblich zu verbessern.

Dies impliziert, dass der entwickelte Algorithmus nur auf einer bestimmten Menge von Modellen arbeiten kann, und dass deshalb mehrere Algorithmen für die verschiedenen in JAMES II verwendeten Modelltypen implementiert werden müssen. Der Nachteil eines höheren Entwicklungsaufwandes wird dabei durch die potentiell höhere Leistungsfähigkeit des Gesamtsystems wettgemacht.

Um die Tragfähigkeit der vorgeschlagenen Lösung unter Beweis zu stellen, soll - neben einem einfachen, allgemeinen Verfahren zur Partitionierung beliebiger Modelle in JAMES II - während der nächsten Kapitel, insbesondere Kapitel 5, ein exemplarischer Algorithmus zur Partitionierung von DEVS-Modellen entwickelt werden, welcher der Einfachheit halber hier schlicht als *DEVS-Algorithmus* bezeichnet wird.

Durch den Anspruch, *mehrere* spezialisierte Partitionierungsalgorithmen integrieren zu können, ergeben sich zwangsläufig neue Anforderungen an die Komponenten, die zur Modellpartitionierung in JAMES II implementiert werden sollen. Diese Anforderungen sind genauer in Abschnitt 3.2.2 beschrieben.

Abschnitt 3.2.1 motiviert dagegen mögliche Anforderungen an einen konkreten Partitionierungsalgorithmus, wobei sich dabei auf den Algorithmus für DEVS-Modelle konzentriert wird. In Abschnitt 3.3 werden alle im nachfolgenden Teil der Arbeit nicht weiter verfolgten Teilaufgaben des beschriebenen Gesamtproblems aufgeführt, denn das praktische Problem besteht nicht nur aus der Partitionierung eines Modellgraphen, sondern wird noch von vielen anderen Faktoren beeinflusst.

Die Abschnitte 3.4 und 3.5 befassen sich schließlich mit den konkreten Designentscheidungen was Entwicklung und Tests der zu erstellenden Komponenten betrifft.

3.2 Anforderungsanalyse

3.2.1 Anforderungen an einen Partitionierungsalgorithmus

Wenn ein theoretisches Problem in der Praxis gelöst werden soll, stellt sich oftmals heraus, dass zusätzliche Anforderungen entstehen, die wegen des Abstraktionsgrads des theoretischen Problems bisher nicht als solche erkannt wurden.

Unglücklicherweise ist dies auch bei diesem Problem der Fall. Betrachtet man die zu lösenden Aufgaben zur Verteilung einer Simulation, stellt man fest, dass die Partitionierung des Modells nur ein Teil der eigentlichen Aufgabe ist: Da das Modell auf eine gewisse Anzahl realer Prozessoren verteilt werden soll, kann nicht davon ausgegangen werden, dass diese homogen sind.

Es ist vielmehr wahrscheinlich, dass sie unterschiedliche Rechenkapazität besitzen (und sei es nur temporär auf Grund externer Prozesse), und dass Bandbreite und Latenzzeit der Netzwerkverbindungen zwischen ihnen - also die Kapazität der Kommunikationsverbindungen - ebenso schwankt. Dieses Problem wird noch verstärkt, wenn man zudem fordert, dass der Partitionierungsmechanismus die Simulation auch auf beliebige Anordnungen von Prozessoren, zum Beispiel auch auf *Grids*, verteilen können soll.

Daher muss die gesamte Infrastruktur in Form eines *Infrastrukturgraphen* in die Berechnungen einer Partition mit eingehen. Im Infrastrukturgraph werden die vorhandenen Prozessoren als Knoten dargestellt und die Netzwerkverbindungen zwischen ihnen als Kanten.

Gesucht wird nun statt der Partitionierung des Modellgraphen an sich eine Abbildung $part : V^M \rightarrow V^I$, eine Abbildung aller Knoten des Modellgraphen $G^M = (V^M, E^M)$ auf die Knoten des Infrastrukturgraphen $G^I = (V^I, E^I)$.

Wie bei Zeigler et al. sind nun die Berechnungskosten der einzelnen Modellelemente von Bedeutung. Damit das Problem so umfassend wie möglich dargestellt werden kann, müssen zusätzlich noch die Kommunikationskosten zwischen zwei Modellelementen, sowie die Berechnungs- und Kommunikationskapazitäten der vorhandenen Infrastruktur mit einbezogen werden:

Sei $cost_{calc} : V^M \rightarrow \mathbb{R}$ eine Funktion, die dem Knoten $v_i^M \in V^M$, der ein Modellelement symbolisiert, die Berechnungskosten dieses Modellelements zuweist. Analog dazu ist $cost_{comm} : V^M \times V^M \rightarrow \mathbb{R}$ eine Funktion zur Ermittlung der Kommunikationskosten zwischen zwei Elementen des Modells, die wiederum durch zwei Knoten im Modellgraph repräsentiert werden.

Genauso kann für die Berechnung der vorhandenen Ressourcen vorgegangen werden: $cap_{calc} : V^I \rightarrow \mathbb{R}$ gibt die Rechenkapazität des mit dem Knoten $v_i^I \in V^I$ assoziierten Prozessors an, und $cap_{comm} : V^I \times V^I \rightarrow \mathbb{R}$ die Kommunikationskapazität zwischen zwei Prozessoren, bzw. deren Knoten im Infrastrukturgraph.

Diese Kostenmaße haben keine Dimension und können, wie in [14] vorgeschlagen, mehrere wichtige Faktoren zur Berechnung zusammenfassen. Beispielsweise sollte die Ermittlung der Kommunikationskapazität zwischen zwei Prozessoren sowohl die Bandbreite der Verbindung als auch deren Latenzzeit in Betracht ziehen.

Außerdem müssen alle Werte relativ interpretiert werden: Ist $cap_{calc}(v_i^I) = 12$ und $cap_{calc}(v_j^I) = 3$, so sagt dies nichts über die tatsächliche Leistung der Prozessoren i und j aus, sondern nur darüber, dass Prozessor i viermal mehr Rechenkapazität als Prozessor j besitzt. Dadurch kann man die mit diesen Maßen

arbeitenden Algorithmen auch in sehr verschiedenen Umgebungen einsetzten.

Definiert man nun eine Funktion $cost'_{calc} : V^I \rightarrow \mathbb{R}$ zur Berechnung der Kosten für alle durch *part* einem Prozessor $v_i^I \in V^I$ zugewiesenen Modellelemente mit

$$cost'_{calc}(v_i^I) = \sum_{v_k^M \in V^M \wedge part(v_k^M)=v_i^I} cost_{calc}(v_k^M)$$

, sowie die Kommunikationskosten zwischen allen durch *part* den Prozessoren $v_i^I, v_j^I \in V^I$ zugewiesenen Modellelementen mit

$$cost'_{comm}(v_i^I, v_j^I) = \sum_{v_k^M \in V^M \wedge part(v_k^M)=v_i^I} \sum_{v_l^M \in V^M \wedge part(v_l^M)=v_j^I} cost_{comm}(v_k^M, v_l^M)$$

, kann das Ziel des Algorithmus sehr einfach ausgedrückt werden. Schließlich geht es eigentlich nur darum, die unausgeglichene Verteilung von Rechenkapazität sowie die Kommunikation zwischen den Prozessoren formal zu beschreiben:

$$imbalance = \sum_{i=1}^{|V^I|} |cap_{calc}(v_i^I) - cost'_{calc}(v_i^I)|$$

$$cutsize = \sum_{i=1}^{|V^I|} \sum_{j=1, j\neq i}^{|V^I|} cost'_{comm}(v_i^I, v_j^I)$$

Hierbei berechnet *imbalance* den Grad des Ungleichgewichts zwischen Auslastung und Kapazität der einzelnen Prozessoren, und *cutsize* berechnet die Menge der benötigten Kommunikation. Um auszudrücken, dass die Kommunikationslast auf guten Netzwerkverbindungen zu bevorzugen ist, könnte statt *cutsize* auch

$$imbalance_{comm} = \sum_{i=1}^{|V^I|} \sum_{j=1, j\neq i}^{|V^I|} \frac{cost'_{comm}(v_i^I, v_j^I)}{cap_{comm}(v_i^I, v_j^I)}$$

definiert werden.

Welche dieser zu minimierenden Funktionen wie stark in einem Algorithmus berücksichtigt werden sollen, hängt stark von den spezifischen Eigenschaften der konkreten Aufgabe ab. Deshalb ist die Definition einer allgemeinen Evaluierungsfunktion hier wenig zweckdienlich.

Mit diesen grundlegenden Gedanken wird nun exemplarisch eine Reihe von konkreten Anforderungen für einen DEVS-Partitionierungsalgorithmus erarbeitet. Abgesehen von der Feststellung, dass neben dem Modellgraphen auch noch die Leistungsfähigkeit und Topologie der vorhandenen Infrastruktur berücksichtigt werden muss, sind die meisten der folgenden Anforderungen auf besondere Charakteristika von DEVS-Modellen zurückzuführen.

Der von Zeigler et al. vorgeschlagene Algorithmus ([14], [15]) besitzt zwar viele positive Eigenschaften, er hat jedoch den großen Nachteil, dass er die Kommunikation zwischen den verschachtelten DEVS-Modellen *nicht* in die Heuristik zur Partitionierung des Modellbaums mit einbezieht. Da der Knoten mit dem

geringsten Berechnungskosten dem am wenigsten gefüllten Partitionsblock zugewiesen wird, bleibt dessen Position innerhalb des Baumes unbeachtet. Dadurch tritt unter Umständen eine starke Fragmentierung des Baumes ein, die wiederum in einer hohen Kommunikationslast bei der Ausführung resultiert. Auch die Topologie der Infrastruktur wurde nicht für den allgemeinen Fall berücksichtigt.

Dies stellt eine zu starke Vereinfachung des Sachverhalts dar. Daher sollte der zu entwickelnde Partitionierungsalgorithmus die Kommunikations- und Berechnungseigenschaften sowohl der Modellelemente als auch der vorhandenen Prozessoren in Betracht ziehen.

Eine weitere Anforderung, die in der Praxis als notwendig erscheint, ist die Möglichkeit, den Algorithmus mit so genannten *Constraints* (engl., Beschränkungen) für die eingeschränkte Suche nach einer geeigneten Partition zu parametrisieren.

Angenommen, es liegt eine Situation wie in Abbildung 3.1 vor, das heißt ein bestimmtes Modellelement muss mit einem externen Prozess, der auf einem festgelegten Prozessor läuft, kommunizieren. Die Definition von Constraints ermöglicht es dem Modellierer, bestimmte Teile des Modells von vornherein bestimmten Prozessoren zuzuweisen. Die Menge der als Lösung möglichen Partitionen wird dadurch also *beschränkt*.

Dies mag zunächst vielleicht abwegig erscheinen, aber gerade bei der Simulation von Multiagentensystemen kann eine ähnliche Situation öfter vorkommen. Das liegt daran, das auch während der Entwicklung eines Multiagentensystems Simulationen verwendet werden, um diese hochkomplexen Systeme auf die gewollten Eigenschaften hin zu überprüfen. Dazu werden bereits vorhandene Komponenten, zum Beispiel Agenten, als externe Prozesse in die Simulation eingebunden. Das simulierte DEVS-Modell beschränkt sich dann auf die Simulation der Umgebung.

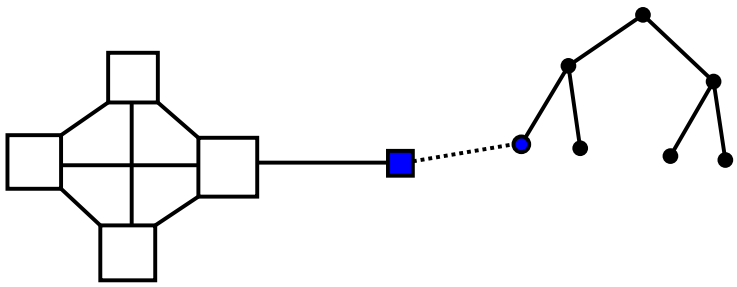

Abbildung 3.1: Hier sieht man einen schlecht verbundenen Prozessor mit einem externen Prozess (blau, links), der beispielsweise über eine *PPP*-Verbindung an das restliche Rechnernetz angebunden wurde. Diesem Prozessor muss das auf diesen Prozess angewiesene Modell zugewiesen werden (rechts, blau). Elternknoten und Geschwister sollten dementsprechend auf dem direkten Nachbarprozessor ausgeführt werden.

Obwohl sich die Forderung nach Beachtung von Constraints recht einfach anhört, hat es sich, wie in Kapitel 5 beschrieben, als sehr schwierig herausgestellt, einen DEVS-Algorithmus zu entwickeln, der halbwegs „intelligent“ mit vorhandenen Constraints umgeht. Bei genauerer Betrachtung ist dies aber kei-

ne Überraschung, schließlich sind so genannte *Constraint Satisfaction Problems*, also Probleme zum Finden einer Lösung in einem beschränkten Suchraum, ein eigenes Forschungsgebiet.

Die für den Partitionierungsalgorithmus definierbaren Constraints lassen sich zusätzlich noch in zwei Gruppen einteilen: Einerseits die oben bereits angesprochenen *konkreten* Constraints, bei denen ein bestimmtes Modellelement einem *konkreten* Prozessor zugeordnet wird.

Andererseits kann aber auch die Festlegung, dass zwei Modellelemente nach der Partitionierung auf dem selben Prozessor ausgeführt werden sollen, eine Rolle spielen. Diese *unbestimmten* Constraints ermöglichen es dem Modellierer, den Algorithmus mit seinem Wissen über das Modell zu unterstützen, indem er beispielsweise Modelle, zwischen denen er intensive Kommunikation erwartet, durch solche Constraints miteinander „verbindet“. Der Vorteil dieser Art von Constraints ist, dass der Modellierer sie bereits während der Modellierung festlegen kann, während konkrete Constraints erst direkt vor Ausführung der Simulation festgelegt werden können. Erst dann sind die Prozessoren, die für die Simulation zur Verfügung stehen, bekannt.

Die Anforderung, dass Constraints dieser Art definierbar sein müssen, beschränkt sich nicht auf Partitionierungsalgorithmen für DEVS, sondern kann auf Algorithmen zur Partitionierung aller Modellarten ausgedehnt werden, sofern diese die Anbindung externer Prozesse erlauben.

Nachdem nun die grundlegenden Anforderungen festgestellt wurden, werden noch einige Annahmen über einen effizienten DEVS-Algorithmus getroffen. Wichtig erscheint bei der Partitionierung von DEVS-Modellen vor allem die Minimierung der Kommunikationskosten, weil bei der Simulation eines DEVS-Modells sehr viele Nachrichten ausgetauscht werden müssen. Der Grund dafür ist die hierarchische Art der Kommunikation. Soll eine Nachricht von einem DEVS-Modell zu einem anderen gesandt werden, kann dies nicht direkt geschehen, vielmehr wird die Nachricht über den entsprechenden Pfad im Modellbaum zum Empfängermodell geleitet.

Der Berechnungsaufwand eines Modells ergibt sich aus den Eigenschaften der atomaren Modelle sowie deren Kopplungen. Bei jedem Fortschreiten der Simulationszeit, signalisiert durch eine * - Nachricht des Root Coordinators an ein atomares Modell, werden zumindest Nachrichten zwischen dem Root Coordinator und dem entsprechenden atomaren Modell ausgetauscht. Soll die Nachricht an andere Modellelemente weitergeleitet werden, geschieht dies wiederum entlang der Kanten des Modellbaums.

Da ein Coordinator ausschließlich der Koordinierung der Simulation dient, werden die generierten Ausgaben meist bis zu anderen Blättern des Modellbaums, also anderen atomaren Modellen, weitergeleitet. Aufgrund der hierarchischen Strukturierung der DEVS-Modelle ist es außerdem wahrscheinlich, dass die Ausgabe eines atomaren Modells vor allem an in der Nähe befindliche atomare Modelle gesandt wird. Mit Nähe ist hier die Existenz eines kurzen Pfades zwischen den Modellen im Modellbaum gemeint.

Aus den Charakteristika der hierarchischen Kommunikation sowie der Ausbreitung von Nachrichten im Modellbaum lässt sich nun folgern, dass die Einteilung des Modellbaums in möglichst große Teilbäume, die nach Möglichkeit erst in der Nähe des Wurzelknotens Kanten zu weiteren Teilbäumen besitzen, von Vorteil ist.

Wenn dies gelingt, müssen weniger Nachrichten zwischen den vorhandenen Prozessoren ausgetauscht werden, da viele interne Zustandsübergänge der atomaren Modelle lediglich lokale Änderungen hervorrufen, die dann intern auf dem Prozessor vorgenommen werden können.

Zusammenfassend ergibt sich aus den prinzipiellen Forderungen an einen Partitionierungsalgorithmus und den hier beschriebenen zusätzlichen Aspekten folgendes Anforderungsprofil für den DEVS-Algorithmus, absteigend nach Anspruch auf Allgemeingültigkeit sortiert:

- Der Algorithmus muss effizient arbeiten und auch für sehr große Modelle gute Partitionen finden
- Der Algorithmus soll gut skalieren, das heißt mit großen Modellbäumen und wenig Prozessoren ähnlich gute Ergebnisse liefern wie mit vielen Prozessoren und kleinen Modellbäumen
- Modellstruktur und Eigenschaften der vorhandenen Infrastruktur müssen beachtet werden
- Konkrete und unbestimmter Constraints müssen so gut wie möglich umgesetzt werden
- Sowohl die Kommunikationskosten als auch das Ungleichgewicht zwischen den Prozessoren müssen minimiert werden, im Falle des DEVS-Algorithmus ist die Reduzierung der Kommunikationskosten von größerer Bedeutung: Daher sollte der Modellbaum in Teilbäume zerlegt werden, die erst möglichst nahe am Wurzelknoten gemeinsame Kanten besitzen.

Die Umsetzung dieser Anforderungen wird anhand der Entwicklung eines konkreten DEVS-Algorithmus in Kapitel 5 veranschaulicht. Zudem wurden noch andere, allgemeine Partitionierungsalgorithmen implementiert, die nur einen Teil der Anforderungen erfüllen. Diese werden in Abschnitt 4.1 genauer erläutert.

3.2.2 Anforderungen an eine Komponente zur Modellpartitionierung

Da die zu entwickelnde Komponente selbst nur ein kleiner Teil des Simulationssystems JAMES II ist, muss sie sich natürlich nahtlos und einfach in das bereits vorhandene System einpassen lassen.

Daher ist es günstig, diese Komponente als *Framework* zu entwickeln. Ein Framework hat zunächst eine gewisse Ähnlichkeit zu einer einfachen Programmbibliothek: Es wird für eine spezielle Teilaufgabe genutzt und bietet eine gewisse Funktionalität, auf die über eine definierte Schnittstelle zugegriffen werden kann.

Im Gegensatz zu einer Programmbibliothek besitzen Frameworks in der Regel extrem kleine Schnittstellen. Die genutzten Methoden zur Lösung des Problems werden also stärker vor dem Nutzer des Frameworks verborgen. Außerdem unterscheiden sich Frameworks von Softwarebibliotheken, indem sie in der Regel sehr leicht erweiterbar sind - wie der Name bereits andeutet, werden von einem Framework vor allem die Rahmenbedingungen für die Nutzung der eigentlichen Algorithmen geschaffen.

In diesem Fall ist die Erweiterbarkeit des Frameworks von besonderer Bedeutung, da es um diverse Algorithmen zur Modellpartitionierung erweiterbar sein soll. Gleichzeitig müssen auch Mechanismen zur Erzeugung der Eingabedaten aus den bereits in JAMES II verfügbaren Objekten vorhanden sein, diese gilt es ins Framework zu integrieren. Die Komponenten zur Erzeugung geeigneter Eingabedaten lassen sich in Komponenten zur Modellanalyse und zur Infrastrukturanalyse aufteilen. Methoden der Modellanalyse erzeugen aus einem JAMES II - Modellobjekt einen Graphen, der dessen interne Struktur widerspiegelt. Analog dazu dient die Infrastrukturanalyse der Erzeugung eines Graphen, in dem die vorhandenen Prozessoren und ihre Verbindungen untereinander abgebildet sind. Beide Bestandteile müssen genauso leicht erweiterbar sein wie die eigentliche Partitionierungskomponente.

Zusätzlich zu diesen allgemeinen Anforderungen ist es möglich, dass das Framework in Zukunft um die nahe liegende Funktion des Load Balancing erweitert werden wird - auch dies muss bei der Gestaltung der Architektur bedacht werden.

3.3 Abgrenzung der Aufgabe

Neben der Vielzahl an Anforderungen, die man an einen Algorithmus zur Partitionierung von Modellen stellen kann, gibt es auch bestimmte Rahmenbedingungen, welche für die korrekte Ausführung des Algorithmus notwendig sind. Besonders wichtig sind dabei die Informationen, die dem Algorithmus über die zur Verfügung stehenden Prozessoren und die einzelnen Elemente des Modells vorliegen.

Leider sind diese Informationen in den meisten Fällen schwer zu ermitteln. Der Berechnungsaufwand für ein Element des Modells kann von Simulation zu Simulation stark variieren, so dass eine Aussage darüber meist nur statistischen Charakter besitzt. Bei JAMES II kommt noch hinzu, dass atomare Modellelemente aus Java-Klassen bestehen, so dass selbst Heuristiken zur Aufwandsabschätzung, wie sie Zeigler et al. vorschlagen, schwer durchführbar sind (siehe Abschnitt 2.5).

Eine ähnliche Situation herrscht bei der Erfassung von Daten über die vorhandenen Prozessoren. Natürlich ist es möglich, die Leistungsmerkmale der Hardware eines Systems, also zum Beispiel Prozessortyp oder die Art und Größe des Hauptspeichers, auszulesen und zu bewerten. Doch damit besäße man nur Informationen über die potentiell vorhandene Rechenkapazität des Prozessors, die tatsächliche momentane Kapazität hängt von vielen weiteren Faktoren ab, etwa der Anzahl und Priorität von konkurrierenden lokalen Prozessen.

Für die Implementierung in JAMES II ist außerdem problematisch, dass die *Java Virtual Machine* (JVM), welche die JAMES II - Komponenten auf dem zu nutzenden Prozessor ausführt, einen weiteren Unsicherheitsfaktor darstellt. Sie läuft auf dem Prozessor als einziger für das Betriebssystem sichtbarer Prozess, in dem dann die Ausführung des JAMES II - Bytecodes vorgenommen wird. Damit sind die Leistungsmerkmale der Hardware hier nur zweitrangig, und statt dessen muss zusätzlich die Speicher- und Rechenkapazität, die der JVM vom Betriebssystem zugeteilt wird, untersucht werden.

Weiterhin gibt es diverse JVM - Versionen von verschiedenen Herstellern, die je nach Betriebssystem und Anwendung mehr oder weniger leistungsstark

sind. Dies alles müsste für eine genaue Abschätzung der vorhandenen Berechnungskapazität berücksichtigt werden. Hinzu kommen ähnliche Probleme bei der Abschätzung von Kommunikationsbedarf zwischen Modellelementen sowie der Kommunikationskapazität zwischen Prozessoren.

Auf das Problem der Graphenpartitionierung bezogen kann man also davon sprechen, dass die Beschriftung der Eingabegraphen, also des Modellgraphen und des Infrastrukturgraphen, jeweils sehr schwer wiegende Probleme darstellen.

Natürlich gibt es verschiedene Möglichkeiten, die gerade beschriebenen Informationsdefizite auszumerzen (siehe Pre-Simulation in Abschnitt 2.7). Es gibt auch - speziell aus dem Bereich der verteilten Systeme und des Grid-Computing - Möglichkeiten, die Hürden bei der Ermittlung der freien Ressourcen eines Prozessors zu überwinden ([18]).

Die Modellanalyse und die Infrastrukturanalyse sind jedoch Probleme, die weit vom ursprünglich formulierten Ziel, der Implementierung eines Systems zur Modellpartitionierung, fort führen, weshalb sie in der konkreten Implementierung durch Triviallösungen (siehe Abschnitt 4.1) ersetzt wurden.

Nichtsdestotrotz ist das Wissen um diese Probleme entscheidend für den Entwurf eines flexiblen und erweiterbaren Partitionierungsframeworks, wie im nächsten Abschnitt beschrieben wird.

3.4 Entwicklung der Framework-Architektur

3.4.1 Entwurf eines geeigneten Gesamtkonzepts

Nachdem die Anforderungen an Partitionierungsalgorithmen formuliert wurden (siehe Abschnitt 3.2.1) und Problemfelder, welche die Leistungsfähigkeit des Algorithmus einschränken können, aufgezeigt wurden (siehe Abschnitt 3.3), sollen diese Überlegungen nun genutzt werden, um die in Abschnitt 3.2.2 formulierten Anforderungen an ein Framework zur Modellpartitionierung zu einem stimmigen Gesamtkonzept zu vereinen.

Es gibt bereits einige vorhandene Ansätze zur Gestaltung eines flexiblen Partitionierungssystems. Diese sind zwar gut durchdacht, aber zu sehr an die speziellen Anforderungen der konkreten Anwendung angepasst, um hier wiederverwendet werden zu können. Ein gutes Beispiel dafür ist die Gestaltung der Architektur eines Modellpartitionierers von Li et al. [21]. Auf Unterschiede und Gemeinsamkeiten zwischen dem vorhanden Entwurf und dem hier vorgestellten wird im Folgenden wiederholt eingegangen.

In der objektorientierten Softwareentwicklung ist die Hinzuziehung von *Software Pattern* (engl., Software - Muster) eine sinnvolle und anerkannte Methode, um die Eigenschaften einer Softwarearchitektur klar herauszustellen. Deshalb wird zunächst die Auswahl bestimmter Muster motiviert, die dann zu einer Gesamtarchitektur komponiert werden.

Modellpartitionierung, die Modellanalyse und die Infrastrukturanalyse stellen jeweils komplexe Probleme dar. Diese können von unterschiedlichen Algorithmen gelöst werden. Damit die Verwendung eines bestimmten Algorithmus für die anderen Teile des Systems transparent bleibt, werden abstrakte Oberklassen für die Algorithmen aus den drei Problembereichen definiert. Mit diesen arbeitet die Klasse `Partitionizer`, welche Modell- und Infrastrukturgraph durch konkrete Unterklassen von `AbstractModelAnalyzer`

bzw. `AbstractInfrastructureAnalyzer` erzeugen lässt, und diese dann an eine konkrete Unterklasse von `AbstractPartitioningAlgorithm` weitergibt. Das Ergebnis - die Partition - wird danach an den Nutzer zurückgegeben, `Partitionizer` ist sozusagen die Schnittstelle zwischen Nutzer und Framework (siehe Abbildung 3.2).

Doch wie werden dem `Partitionizer` die jeweils besten Algorithmen für die entsprechende Aufgabe zugänglich gemacht, und wie kann eine Auswahl unter den vorhandenen Algorithmen vorgenommen werden? Dieses Problem, also die Auswahl zwischen verschiedenen zur Instantiierung in Frage kommenden Klassen zur Laufzeit, kann bei Objektorientierten Programmiersprachen durch das so genannte *Factory* - Pattern gelöst werden.

Diese Metapher ist darin begründet, dass hier eine Fabrik-Klasse definiert wird, die zur Laufzeit bestimmte Objekte herstellen soll. Die Fabrik-Klasse nutzt dabei die *Polymorphie* von objektorientierten Sprachen aus, indem sie selbst entscheidet, welche Unterklasse des Rückgabetyps instantiiert wird. Damit produziert eine Fabrik - in Analogie zur physikalisch vorhandenen Fabrik - Objekte eines festgelegten Typs, die Interna der Produktion sind dem aufrufenden Objekt jedoch verborgen.

Eine einzelne Factory bietet jedoch nicht die volle Flexibilität, die im hier besprochenen Fall benötigt wird: Es gibt nicht nur die verschiedensten Ansätze, wie oben besprochenen Probleme zu lösen sind, auch die *Auswahl* eines geeigneten Algorithmus kann durch sehr unterschiedliche Kriterien erfolgen: So ist es denkbar, dass eine Herangehensweise zur Graphenpartitionierung Informationen über die verschiedenen Modellierungssprachen ausnutzt, und deshalb je nach Modellierungsformalismus einen passenden Algorithmus aussucht.

Eine andere Herangehensweise könnte die Eingabedaten jedoch ganz anders unterteilen, beispielsweise nach Größe des eingegebenen Modells („Nutze schnellere Algorithmen, je größer die Modellstruktur ist.“) oder auch nach der Art der gegebenen Infrastruktur (Algorithmen für Cluster, Grids, etc.). Die gleichen Überlegungen lassen sich auch für Factory-Klassen zur Erzeugung von Algorithmen zur Modell- und Infrastrukturanalyse vollziehen, denn auch hier lässt sich ein Algorithmus nach sehr verschiedenen Gesichtspunkten auswählen.

Generell wird also eine doppelte Flexibilität gefordert: Zum einen die Möglichkeit, problemlos neue Algorithmen für jedes einzelne Problem hinzuzufügen, ohne dass andere Teile des Systems davon betroffen sind. Zum anderen müssen auch die *Auswahlmechanismen*, also die verschiedenen Factories, austauschbar sein, um verschiedene Zuteilungsstrategien einsetzen zu können.

Zur Lösung dieses Problems wird hier das *Abstract Factory* - Pattern [19, S. 87 - 96] genutzt, da es für die geschilderten Anforderungen formuliert wurde. Dieses Pattern wird für jeden der drei Problembereiche des Algorithmus angewendet, wobei die dafür notwendigen Klassen zur besseren Übersicht in eigene Pakete einsortiert wurden.

Beim Abstract Factory-Pattern wird die Art der Abstraktion, die schon für die Erzeugung von verschiedenen Algorithmus-Objekten verwendet wurde, auf die Ebene der Factory-Objekte ausgedehnt. Das Nutzer-Objekt muss nun zuerst die statische Funktion `newInstance()` einer zusätzlichen Factory-Klasse aufrufen. Diese Methode instantiiert dann ein Factory-Objekt und gibt es zurück.

Damit ist die zusätzliche Factory-Klasse eine Factory für Factories, sozusagen eine Meta-Factory. Hier kann man definieren, welche Art der Algorithmenwahl, also welche Factory-Klasse, der übergeordneten `Partitionizer`-Klasse

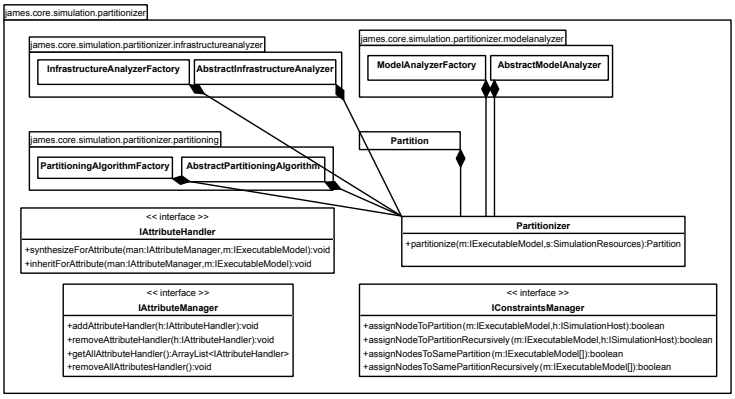

Abbildung 3.2: Hier sieht man deutlich, dass nur die Klasse `Partitionizer` auf die in den Unterpaketen definierten Klassen zurückgreift. Sie bildet die Schnittstelle des Frameworks

zur Verfügung gestellt wird.

Die Umsetzung dieses Patterns ist in Abbildung 3.3 zu sehen, die Pakete für die Infrastruktur- und Modellanalyse sind analog aufgebaut. Entsprechende UML-Diagramme befinden sich im Anhang A.

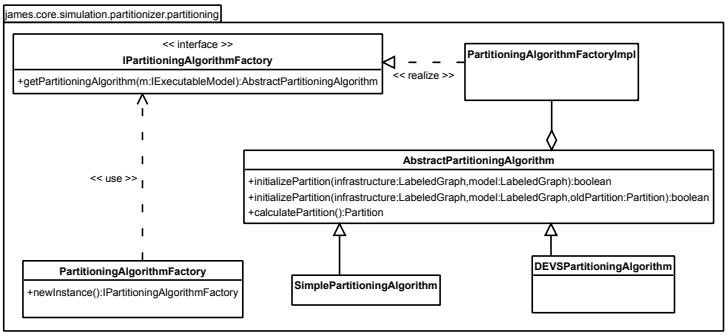

Abbildung 3.3: Aufbau des Frameworks für Partitionierungsalgorithmen. `PartitioningAlgorithmFactory.newInstance()` wird aufgerufen und gibt die Instanz einer Factory - Klasse, zum Beispiel `PartitioningAlgorithmFactoryImpl`, zurück. Diese instantiiert dann beim Aufruf von `getPartitioningAlgorithm(...)` einen passenden Algorithmus, in diesem Fall beispielsweise `SimplePartitioningAlgorithm` oder `DEVSPArtitioningAlgorithm`.

Li et al. beschreiben in [21] die Architektur eines Partitionierers zur Simulation von Modellen für Computerhardware. Um Algorithmen miteinander vergleichen zu können, wird auch hier eine abstrakte Oberklasse für Partitionierungsalgorithmen erstellt. Die *Auswahl* eines Algorithmus geschieht hier jedoch nur halbautomatisch, das heißt hier wird der genutzte Algorithmus zur Laufzeit anhand der beim Programmstart in die Kommandozeile eingegebenen

Parameter ausgewählt. Dies ist sicherlich ausreichend, wenn man die Art und Beschaffenheit der zu partitionierenden Modelle abschätzen kann, in diesem Fall handelt es sich wie gesagt um Modelle für Hardware.

Da mit JAMES II jedoch Modelle aus den unterschiedlichsten Bereichen wie Systembiologie, Soziologie, Robotik usw. verteilt ausgeführt werden sollen, die Struktur der Modelle also völlig unbekannt ist, verspricht eine Verallgemeinerung des Ansatzes von Li et al. hier den Vorteil einer vereinfachten Integration von gänzlich neuen Algorithmen*mengen*, deren Auswahlfunktion zur Entwurfszeit noch nicht feststeht.

Weiterhin werden beim Entwurf von Li et al. alle Informationen zum Aufbau des Modells durch einen Parser für die verwendete Hardwarebeschreibungssprache, also dem verwendeten Modellierungsformalismus, generiert. Dies ist auch ein Beispiel für eine optimal an die konkrete Aufgabe angepasste Lösung, die dadurch jedoch der für JAMES II essentiellen Forderung nach Flexibilität widerspricht: Da das von Li et al. vorgestellte Simulationssystem *ausschließlich* Hardwaremodelle in einem Formalismus simulieren soll, ist die Herangehensweise über den Parser folgerichtig und effizient, zumal der Parser durch Unterklassen von ihm ersetzt werden kann, die Aspekte verschiedener Simulatoren berücksichtigen.

Im hier vorliegenden Fall sollen prinzipiell *verschiedene* Modellierungsformalismen unterstützt werden, so dass die Analyse eines Modells in einem flexibleren Rahmen stattfinden muss.

Nachdem die Architektur des Frameworks nun feststeht, wird in den nächsten Abschnitten auf verschiedene Besonderheiten beim Entwurf eingegangen.

3.4.2 Attributierung von Modellelementen

Dieser Abschnitt beschäftigt sich mit einem Mechanismus, der es externen Objekten erlaubt, während der Modellanalyse eigene Daten zu erheben, und diese bei Bedarf als Attribute der Modellelemente zu speichern.

Die Attributierung von Modellelementen ist dann sinnvoll, wenn schon bei der Modellpartitionierung Informationen ermittelt werden können, die für das Erzeugen geeigneter Simulator-Objekte etc. benötigt werden. Dies läuft natürlich in gewisser Weise der Bestrebung zuwider, die Modellpartitionierung und die darauf folgende Ausführung der Simulation so weit wie möglich zu entkoppeln.

Andererseits ist durch die Attributierung eine Vereinfachung der Factory - Klassen für die Simulator-Objekte möglich, da diese auf schon vorhandene Informationen zurückgreifen können. Außerdem kann die Leistungsfähigkeit des Systems erhöht werden, weil das Modell nur einmal komplett analysiert werden muss und nicht öfter.

Das Problem der dadurch entstehenden Abhängigkeit der Factory-Klassen für Simulator-Objekte von Algorithmen, die diese Attributierung unterstützen, kann durch den Einsatz von geeigneten Auswahlmechanismen gelöst werden.

Da die Attributierung von einzelnen Elementen aber nur bei gewissen Modellierungsformalismen, wie zum Beispiel DEVS, wünschenswert ist, muss eine separate Schnittstelle dafür geschaffen werden, die nicht von der abstrakten Oberklasse der Partitionierungsalgorithmen implementiert wird, sondern nur von konkreten Algorithmen. Diese können dem Programmierer die Möglichkeit

zur Attributierung als ein zusätzliches Feature zur Verfügung stellen. Dadurch kann auch zur Laufzeit festgestellt werden, ob ein Algorithmus die Attributierung der Modellelemente unterstützt oder nicht.

Dieses Feature bei einem Partitionierungsalgorithmus anzubieten ist nur dann sinnvoll, wenn der Algorithmus ohnehin eine zusätzliche Analyse des vom Modellanalysealgorithmus generierten Modellgraphen durchführen muss. Dies könnte beispielsweise bei DEVS-Modellen der Fall sein, weil deren Modellgraph ein Baum ist und es somit von Vorteil sein kann, bestimmte Informationen zu erheben und mit den Knoten zu assoziieren.

Doch wie erfährt ein Partitionierungsalgorithmus, welche Attribute zum Beispiel von der Factory-Klasse für Simulator-Objekte des entsprechenden Modellierungsformalismus benötigt werden? Und wie können andere JAMES II - Komponenten ebenfalls von der Möglichkeit der automatischen Attributierung profitieren?

Um dieses Problem zu lösen, wird auf das so genannte *Observer-Pattern* [20, S. 209-217] zurückgegriffen. Hierbei handelt es sich um ein Pattern, dass die Kommunikation zwischen einem Objekt, hier das Objekt des Partitionierungsalgorithmus, und einer Menge von Objekten, die es „beobachten“, regelt. Die Beobachter (engl.: observer) können dabei auf gewisse Ereignisse reagieren, die das beobachtete Objekt auslöst. In diesem konkreten Fall ruft der Partitionierungsalgorithmus für jeden Knoten im Modellgraphen, den er zum ersten Mal analysiert, alle registrierten Beobachter auf, so dass diese genau die Attribute ermitteln können, die für sie selbst von Bedeutung sind.

Wie in Abbildung 3.2 zu sehen, wurden zur Umsetzung dieses Patterns zwei Schnittstellen definiert. Ein Partitionierungsalgorithmus, der eine Attributierung der Elemente vornehmen kann, muss die Schnittstelle `IAttributeManager` implementieren. Diese definiert Methoden zur Registrierung von Beobachter-Objekten, sowie eine Reihe von Funktionen, welche die Beobachter-Objekte im Falle einer Benachrichtigung benötigen könnten. Letztgenannte wurden wegen ihrer großen Anzahl nicht zum Diagramm hinzugefügt.

Die zweite Schnittstelle, `IAttributeHandler`, ist die Schnittstelle, die Beobachter-Objekte implementieren müssen. Hier gibt es zwei Funktionen, `synthesizeForAttribute(...)` und `inheritForAttribute(...)`. Wie ist dies zu erklären, da doch eigentlich nur ein Ereignis, nämlich die Analyse eines Elements, beobachtet werden soll?

Dies ist als eine bessere Unterstützung von hierarchischen Modellierungsformalismen wie DEVS zu verstehen, deren Modellgraphen Bäume sind. In der Übersetzertechnik werden so genannte *attributierte Grammatiken* [22] zur Dekoration eines abstrakten Syntaxbaumes mit für die Generierung des Maschinencodes wichtigen Informationen benutzt.

Grob gesagt sind attributierte Grammatiken kontextfreie Grammatiken, wobei zu jedem Element durch zu den Erzeugungsregeln gehörige Berechnungsvorschriften eine Menge von Attributen definiert werden kann. Dabei wird zwischen *synthetisierten* und *vererbten* Attributen unterschieden: Die synthetisierten Attribute eines Elements werden durch Attribute von den Elementen berechnet, die aus dem aktuellen Element erzeugt werden. Diese Elemente werden im daher *Ableitungsbaum* eines Wortes als Kindknoten des Elements dargestellt.

Ererbte Attribute werden dagegen aus Attributen des Elements berechnet, aus dem das aktuelle Element erzeugt wurde, also dem Elternknoten im Ableitungsbaum. Bildlich ausgedrückt werden die synthetisierten Attribute von unten

nach oben und die ererbten Attribute von oben nach unten im Ableitungsbaum berechnet. Eine Illustration dieses Sachverhalts stellt Abbildung 3.4 dar.

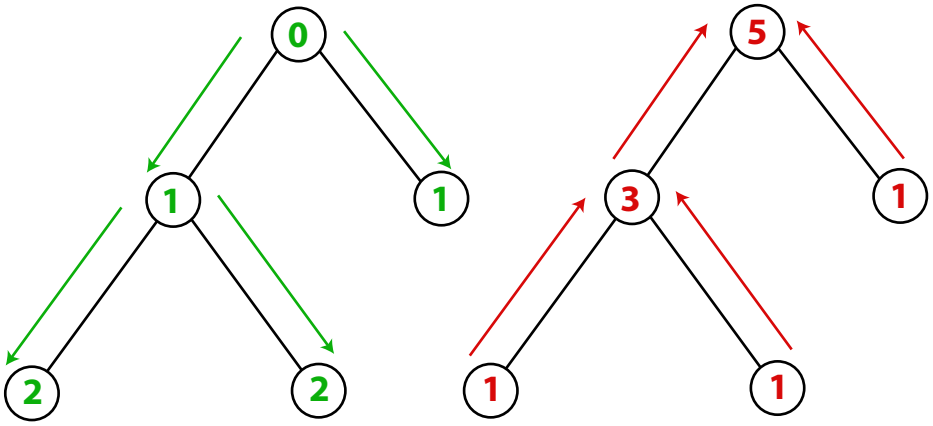

Abbildung 3.4: Auf der linken Seite ist die Zuweisungsreihenfolge des vererbten Attributs 'Tiefe' abgebildet, auf der rechten Seite ist dieser Vorgang für das synthetisierte Attribut 'Anzahl der Knoten im Teilbaum' dargestellt. Wie man sieht, werden vererbte Attribute (grün) *top-down*, und synthetisierte Attribute (rot) *bottom-up* zugewiesen. Die Pfeile sollen die Datenabhängigkeiten verdeutlichen.

Damit gibt es eine bereits entwickelte Methode zum Attributieren von Knoten eines Baumes. An diese Methode angelehnt wird die Funktionsweise der Schnittstelle `IAttributeHandler` für die Beobachter-Objekte hier definiert: Die Methode `inheritForAttributes(...)` wird immer aufgerufen, *bevor* die Kinder des Knotens analysiert wurden, und die Methode `synthesizeForAttributes(...)` immer dann, wenn bereits alle Attribute der Kinder festgelegt sind.

So könnte man die Tiefe eines Knotens im Modellbaum als ererbtes Attribut definieren, mit $depth(v_{root}) = 0$ und $depth(v_i) = depth(v_{parent}) + 1$. Es gilt $v_i \neq v_{root}$ und v_{parent} ist der Elternknoten von v_i. Analog wären die Gesamtkosten des darunter liegenden Teilbaums ein synthetisiertes Attribut, dass für jeden Knoten aus seinen Berechnungskosten, addiert zu den Gesamtkosten zur Berechnung seiner Kinder, ermittelt werden kann.

Es ist leicht zu erkennen, dass auf diese Weise sehr schnell und effizient selbst sehr komplexe Attribute definiert und berechnet werden können. „Echte“ attributierte Grammatiken bilden eine theoretische Grundlage für spezielle Aspekte beim Bau von Übersetzern, hier wird also nur in *Anlehnung* daran ein Verfahren gewählt, dass den Entwicklern erlaubt, schnell und elegant eine große Menge von Attributen zu berechnen.

Tatsächlich erhalten die Beobachter-Objekte für die Berechnung eines Attributs zusätzlich noch eine Referenz auf den `IAttributeManager` (siehe Abbildung 3.2), so dass sie mit Hilfe seiner Funktionen zur Ermittlung von Modellelementen wesentlich mehr Möglichkeiten zur Attributberechnung besitzen als attributierte Grammatiken formal zulassen.

Zwar kann man einwenden, dass nicht alle Formalismen die für diese Vorge-

hensweise nötige Baumstruktur besitzen, jedoch kann zur Analyse der Knoten eines zusammenhängenden Graphen G immer ein *Spannbaum*, also ein kreisfreier Teilgraph, der alle Knoten von G enthält, konstruiert werden.

Auf einem Spannbaum wäre das Verfahren dann wiederum anwendbar. Natürlich ist es auch möglich, dass die eigentliche Herangehensweise, also die des rekursiven Abstiegs in ein Modell, nicht der Natur des Formalismus entspricht. Zelluläre Automaten sind ein Beispiel für einen Modellierungsformalismus, der an sich keine Hierarchie besitzt, und bei dem deshalb auch die Konstruktion eines Spannbaums wenig anschaulich ist.

In solchen Fällen können jedoch auch die Attribute, die den Modellelementen zugeordnet werden sollen, keinerlei Bezug auf Hierarchien besitzen, und der entsprechende Partitionierungsalgorithmus kann ohne semantische Probleme beide Methoden von `IAttributeHandler` hintereinander aufrufen.

Somit ist diese Verfeinerung der Schnittstelle von einer auf zwei Methoden eine Erleichterung der Definition von Attributen bei hierarchischen Modellierungsformalismen wie DEVS, aber sie ist nicht hinderlich, wenn diese Herangehensweise ungünstig oder unmöglich sein sollte.

Auch die konkrete Verwaltung der Attribute in Datentypen kann vom Modellierungsformalismus abhängig sein. Bei der Beispielimplementierung für DEVS-Modelle werden sie in einem Objekt vom Typ `java.util.HashMap<String, Object>` gespeichert, so dass einerseits Eindeutigkeit bezüglich des Namens bei den Attributen herrschen muss (die allerdings mit dem vollständigen Klassennamen als Präfix oder ähnlichen Ansätzen leicht gesichert werden kann), und andererseits auch Attribute möglich sind, die beliebige Objekte als Wert annehmen können.

Wozu nun dieser ganze Aufwand? Bei der Erzeugung von Koordinatoren und Simulatoren zur Simulation eines DEVS-Modells ist beispielsweise zu beachten, welche Modelle an externe Prozesse gekoppelt sind, welche Teile dynamische Strukturänderungen zulassen sollen oder welche davon sequentiell ausgeführt werden können. Diese Details ließen sich schon während der Partitionierung des Modells feststellen. Damit könnte die Entwicklung von entsprechenden Factory-Klassen für die Simulator-Objekte signifikant vereinfacht werden.

3.4.3 Definition von Constraints

Im Gegensatz zur Attributierung von Modellelementen stellt die Unterstützung von Partitionierungsalgorithmen, die man mit Constraints parametrisieren kann, keine große Schwierigkeit dar. Auch hier kann - analog zur Attributierung von Modellelementen - nicht davon ausgegangen werden, dass jeder mögliche Partitionierungsalgorithmus zwangsläufig dieses Merkmal besitzen muss. Daher wird auch hier auf eine spezielle Schnittstelle, `IConstraintsManager`, zurückgegriffen, die ein konkreter Algorithmus implementieren kann, wenn er diese Funktionalität anbieten will.

Anders als bei der Attributierung ist hier aber kein aufwendiges Kommunikationspattern notwendig. Es genügt, wenn der Nutzer über eine Menge von Funktionen die vorhandenen Constraints an den Algorithmus weitergeben kann.

Wie in Abschnitt 3.2.1 festgestellt, gibt es eigentlich nur zwei Arten von Constraints, die für einen Partitionierungsalgorithmus in Frage kommen: Zum einen konkrete Constraints, also Zuweisungen von bestimmten Modellelementen zu bestimmten Prozessoren. Zum anderen sollte eine Definition unbestimmter

Constraints möglich sein, die es dem Nutzer erlauben, Elemente des Modells dem gleichen, jedoch unbestimmten, Partitionsblock zuzuordnen.

Für beide Arten der Constraintsfestlegung gibt es jeweils eine Grundfunktion, `assignNodeToPartition(...)` für konkrete und `assignNodesToSamePartition(...)` für unbestimmte Constraints. Zur einfacheren Handhabung wird zu jeder Funktion noch eine rekursive Funktion mit dem Suffix '`Recursively`' definiert, welche die angegebenen Constraints nicht nur mit den einzelnen Modellelementen, sondern auch mit diesen untergeordneten Teilen des Modells assoziiert, falls das Modell hierarchisch aufgebaut ist.

Jede Funktion erzeugt eine `ConstraintConflictException`, falls das angegebene Constraint nicht erfüllt werden kann. Ordnet man ein Modellelement beispielsweise zwei verschiedenen Prozessoren fest zu, so ist mindestens eines der Constraints *nicht* erfüllbar, und der zweite Funktionsaufruf sollte dies dem aufrufenden Objekt mit einer `ConstraintConflictException` signalisieren. Diese enthält zudem den Index des ersten Knotens, dessen Betrachtung die Hinzuziehung des Constraints zum Scheitern gebracht hat. Damit ist es möglich, dem Nutzer eine verständliche Fehlermeldung zu präsentieren, oder sogar automatisch auf den Fehler zu reagieren.

Damit der Partitionierungsalgorithmus die über die Schnittstelle definierten Constraints in diesem Sinne korrekt verarbeiten kann, muss er zu diesem Zeitpunkt bereits eine Referenz zum Modellgraph besitzen oder ihn gegebenenfalls in eine interne Darstellung umgewandelt haben. Andernfalls gäbe es keine Möglichkeit, die Erfüllbarkeit von Constraints zu überprüfen, da die Struktur des Modellgraphen für die Kontrolle der mit den rekursiven Funktionen definierten Constraints benötigt wird.

Deshalb wird die Schnittstelle des abstrakten Partitionierungsalgorithmus (siehe Abbildung 3.3) so gewählt, dass seine Initialisierung und Ausführung über zwei getrennte Methoden veranlasst werden: Zunächst muss `initializePartition(...)` aufgerufen werden, erst danach kann der Algorithmus mit Constraints parametrisiert werden. Mit `calculatePartition()` wird dann der eigentliche Partitionierungsprozess vom `Partitionizer`-Objekt gestartet.

3.4.4 Erweiterbarkeit hinsichtlich Load Balancing

In Abschnitt 3.2.2 wurde gefordert, dass auch auf die generellen Anforderungen einer Architektur für Load Balancing Rücksicht genommen werden muss. Zwar bleiben Load Balancing - Verfahren bei dieser Arbeit außen vor, das Framework für die Modellpartitionierung sollte aber so gestaltet werden, dass einer nahtlosen Einbindung von Load Balancing - Verfahren nichts im Weg steht.

Da die gesamte Architektur, was die Auswahl und Gestaltung der verschiedenen Arbeitsschritte anbelangt, voll flexibel ist, sollte eine Integration von entsprechenden Factories und Algorithmen kein Problem sein.

Beispielsweise könnte eine entsprechende Factory - Klasse prüfen, ob die Simulation bereits ausgeführt wird. Wenn dem so ist, werden an Stelle der für die Modellpartitionierung etc. verwendeten Algorithmen Verfahren genutzt, die an das Load Balancing - Problem angepasst sind. Zusätzliche Informationen über den Verlauf der Simulation, die Load Balancing - Verfahren unter Umständen benötigen, könnten diese in statischen Klassenvariablen speichern, so dass eine

wiederholte Instantiierung der Algorithmen durch die Factories keine Auswirkungen auf die Datenkonsistenz hat.

Um die Repartitionierungsalgorithmen des Load Balancings so kompatibel wie möglich zu den „normalen“ Partitionierungsalgorithmen zu halten, wurde die Definition des `AbstractPartitioningAlgorithm` um die Initialisierungsfunktion `initializePartition(..., Partition oldPartition)` erweitert. So kann jeder Partitionierungsalgorithmus zwei verschiedene Verfahren in sich kapseln: Eines für die initiale Partitionierung, und eines, das über ein bereits berechnetes Ergebnis verfügt, zum Load Balancing.

3.5 Erarbeitung von Testkriterien

Damit sich die Leistungsfähigkeit der verschiedenen Komponenten objektiv bewerten lässt, müssen zusätzlich zu den entwickelten Komponenten weitere Mechanismen erarbeitet werden, um diese zu testen.

Höchste Priorität besitzt dabei eine ansatzweise Validierung der Komponenten, indem ihre Ausgabe für verschiedenste Eingaben mit den an sie gestellten Erwartungen verglichen wird.

Natürlich kann damit keine formal begründbare Aussage über die korrekte Funktionsweise des Systems getroffen werden. Es ist jedoch nahe liegend, dass durch diese Vorgehensweise die Wahrscheinlichkeit, einen sporadisch oder nur bei Sonderfällen auftretenden Fehler zu finden, drastisch erhöht werden kann.

Um aber effektiv nach Fehlern aller Art zu suchen, ist es ratsam, möglichst viele verschiedene Eingabefälle heranzuziehen. Eine einfache Möglichkeit, die benötigte repräsentative Menge an Eingabefällen zu erlangen, ist die stochastische Generierung von Eingabedaten, das heißt Hardware- und Modellgraphen. Dies ermöglicht, zusammen mit Algorithmen zur Kontrolle und Bewertung der Ergebnisse, eine vollständige Automatisierung des Testvorgangs.

Eine weitere Motivation für diesen Schritt besteht darin, dass sich so die Leistungsmerkmale der entwickelten Komponente abschätzen lassen. Außerdem kann der Einfluss von Constraints oder anderen Eigenschaften der Eingabedaten auf die Qualität der Ausgabe ermittelt werden. Die Eingabedaten für die Testfälle werden durch folgende Merkmale charakterisiert:

- Größe des Infrastrukturgraphen
- Größe des Modellgraphen
- Anzahl an zufällig erzeugten Constraints. Dabei wird sich im Folgenden auf konkrete Constraints beschränkt, da eine Unterscheidung für den DEVS - Algorithmus keine Rolle spielt (siehe Kapitel 5) und kein anderer Algorithmus Constraints unterstützt.
- Durchschnittliche Anzahl der Kinder bzw. Nachbarn im Modellbaum bzw. -graph, diese gibt Aufschluss über die Struktur des Modellgraphs

Die Leistung eines Algorithmus lässt sich dann mit der Untersuchung der erzielten Ergebnisse bezüglich der Kriterien

- Schnittgröße, das heißt das Ausmaß der Kommunikation zwischen den verschiedenen Partitionsblöcken (siehe Abschnitt 3.2.1, *cutsize*)

- Ungleichgewicht zwischen den verschiedenen Partitionsblöcken (siehe Abschnitt 3.2.1, *imbalance*)
- Benötigte Zeitdauer zur Partitionierung

einschätzen. Nun kann man für jeden verwendeten Partitionierungsalgorithmus prüfen, inwiefern sich Veränderungen der Eingabedatenmerkmale auf die Leistung des Algorithmus auswirken - dies ergibt eine Vielzahl sehr interessanter Testszenarien.

Anhand der durch diese Tests gewonnenen Informationen ist es eventuell möglich, in Zukunft konkrete Faustregeln zur Nutzung des Frameworks in der Praxis aufzustellen, beispielsweise wie viele Constraints maximal eingegeben werden sollten, wenn man noch eine spürbare Beschleunigung der Simulation durch die verteilte Ausführung erreichen möchte.

Dass durch diese Gestaltung des Testvorgangs nur die Funktionsfähigkeit der Partitionierungsalgorithmen überprüft wird, ist zumindest für das Spektrum dieser Arbeit nicht problematisch, da die verwendeten Modellanalyse- und Infrastrukturanalysealgorithmen trivial genug sind, um sie mit herkömmlichen Mitteln ausreichend zu evaluieren (siehe Abschnitt 4.1). In Abschnitt 4.3 ist die technische Realisierung der Testkomponenten beschrieben und in Kapitel 6 werden die durchgeführten Tests genauer erläutert, sowie deren Ergebnisse ausgewertet.

Kapitel 4

Details zur Umsetzung

4.1 Implementierte Algorithmen

Um das in Kapitel 3 entworfene System auch nutzen zu können, muss eine Implementierung zu jedem der drei Arbeitsschritte - Modellanalyse, Infrastrukturanalyse und Partitionierung - mindestens einen funktionsfähigen Algorithmus enthalten.

Wie in Abschnitt 3.3 begründet, wird dabei für die Analyse der Modelle bzw. der Infrastruktur jeweils eine „Triviallösung“ verwendet. So wird angenommen, dass alle vorhandenen Prozessoren die gleiche Rechenkapazität besitzen, und dass jeder Prozessor mit jedem anderen eine gleich gute Netzwerkverbindung besitzt. Es wird also für n Prozessoren ein kompletter Graph mit n Knoten konstruiert, dessen Kanten- und Knotenbeschriftungen alle 1 sind. Dieses Vorgehen wurde in der Klasse `SimpleInfrastructureAnalyzer` implementiert.

Bei der Konstruktion des Modellgraphen wird ähnlich vorgegangen, allerdings ist hier die Struktur durch den Formalismus bereits vorgegeben. Da bisher vor allem DEVS-Modelle mit JAMES II simuliert wurden, wurde hier lediglich eine Klasse zur Analyse von DEVS-Modellen implementiert. Hierbei wird ein Graph erzeugt, dessen Struktur der des Modellbaums entspricht. Der Berechnungsaufwand wird für alle Elemente des Modells als gleich angesehen, daher werden alle Knoten auch hier mit 1 beschriftet.

Weiterhin wird angenommen, dass der Kommunikationsbedarf eines Modells proportional zur Anzahl der für dieses Modell definierten Kopplungen ist. Daher wird die Kante zwischen einem Modell und seinem darüber liegenden gekoppelten Modell mit der Anzahl der Kopplungen, die für das Modell definiert sind, beschriftet. Die Klasse `SimpleDEVSAnalyzer` implementiert diese Vorgehensweise.

Da mit dem in Kapitel 5 entwickelten DEVS-Algorithmus nur DEVS-Modelle partitioniert werden können, wurde zusätzlich ein möglichst stabiles und einfaches Verfahren zum Partitionieren von beliebigen (nicht-DEVS-)Modellen implementiert, nämlich das KL-Verfahren (siehe Abschnitt 2.2). Es ist in der Klasse `KLPartitioningAlgorithm` zu finden.

Leider liegt die Zeitkomplexität des KL-Verfahrens in $O(n^3)$, so dass es sich nicht gut für besonders große Graphen eignet. Außerdem arbeitet es auf Matrizen, dass heißt der Speicherplatzbedarf ist ebenfalls polynomiell, er liegt in

$O(n^2)$. Ein Graph mit 10.000 Knoten würde damit eine Matrix mit 100.000.000 Elementen benötigen. Also selbst wenn man nur ein Byte zum Speichern eines Wertes verwenden würde, benötigte allein eine Matrix dieser Größe ca. 800 MB Arbeitsspeicher.

Natürlich handelt es sich dabei meist um *schwach besetzte Matrizen*, und man kann spezielle Bibliotheken wie das *Java Sparse Matrix Toolkit* [35] verwenden, um die Größe der Matrizen im Speicher zu reduzieren. Das Problem der hohen Zeitkomplexität lässt sich leider nicht so einfach lösen, obwohl es zahlreiche Optimierungsvarianten für verschiedene Details des Algorithmus gibt.

Deswegen wurde ein weiteres Verfahren implementiert, das vor allem *schnell* partitioniert. Die Basis für das zusätzliche Verfahren stellt ein geometrischer Ansatz dar [1], der im Grunde nur eine Liste von Knoten des Modellgraphen auf eine vorgegebene Anzahl Blöcke aufteilt. Interessant ist vor allem, wie diese Liste von Knoten ermittelt wird, schließlich besitzen die Knoten in diesem Fall keinerlei geometrische Semantik, die man dafür hätte verwenden können. Statt dessen wird ein modifiziertes *Cuthill-McKee - Verfahren* [25] genutzt, das ursprünglich für die Bandbreitenverringerung in schwach besetzten Matrizen zum Einsatz kam. Der Algorithmus ist daher eine Abwandlung der *Index Based Partitioning* - Methode (siehe Abschnitt 2.4).

Die Grundidee des Cuthill-McKee - Verfahrens wird nun auf die Adjazenzmatrix des Modellgraphen bezogen. Aus der graphentheoretischen Perspektive führt der Algorithmus eine Umnummerierung der Knoten aus, so dass benachbarte Knoten möglichst nahe beieinander liegende Nummern erhalten.

Dafür wird in diesem Fall zunächst der Knoten mit den geringsten Kommunikationskosten gewählt, er bildet das erste Element einer Liste. Dann werden alle Nachbarn des Knotens in eine Liste kopiert, welche aufsteigend nach Kommunikationskosten sortiert wird. Diese Liste wird danach an die Liste mit dem ersten Knoten angehängt. Dann wird der nächste Knoten der neuen Liste ausgewählt (also der Nachbar des ersten Knotens, der die geringsten Kommunikationskosten besitzt), und alle noch nicht in der Liste befindlichen Nachbarn dieses Knotens werden wiederum nach Kommunikationskosten sortiert und der Liste hinzugefügt. Dann wird der nächste Knoten der Liste gewählt, usw..

Dies geschieht so lange, bis die Liste alle Knoten des Graphen enthält. Der Kerngedanke bei diesem Algorithmus ist also, Knoten mit hohem Kommunikationsaufwand weiter hinten anzuordnen, da sie wahrscheinlich Kanten zu mehreren noch nicht in der Liste befindlichen Knoten besitzen, und auch zu diesen Knoten möglichst nahe angeordnet sein sollten. Es gibt noch eine Variation des Verfahrens, bei dem die Kanten zu bereits in der Liste befindlichen Knoten bei der Kostenberechnung ignoriert werden. Diese Verfeinerung wurde hier vermieden um Zeit zu sparen. Insgesamt verbraucht der Algorithmus sehr wenig Speicherplatz und arbeitet sehr schnell.

Nichtsdestotrotz wird auch das KL-Verfahren noch im Framework verwendet, allerdings nur für kleine Graphen mit weniger als 100 Knoten, weshalb es beim Testen der Leistungsfähigkeit der Algorithmen (siehe Kapitel 6) außen vor gelassen wurde.

4.2 Repräsentation von Graphen

Da sowohl die zu partitionierenden Modellelemente als auch die Infrastruktur, auf denen sie simuliert werden sollen, als Graphen dargestellt werden, ist eine effiziente und bedarfsgerechte Entwicklung von Klassen zur Repräsentation von Graphen essentiell für eine gute Leistung des Gesamtsystems.

Nun stellt sich die Frage, wie man die als Datenstruktur benötigten Graphen am besten implementiert. Denkbar sind viele Varianten, beispielsweise die Implementierung von Knoten als Klassen, die dann Referenzen zu anderen Knoten besitzen, um so die Kantenmenge darzustellen.

Li et al. [21] nutzen für ihren Entwurf eines objektorientierten Partitionierers eine Graphenklasse, die eine Menge von Kanten- und Knotenobjekten besitzt. Um die verschiedenen Elemente der Hardwaremodelle mit den Elementen des Graphen zu verbinden, werden für diese Unterklassen der Knotenklasse definiert. Das heißt, die Modellelemente selbst bilden einen Graphen, indem sie von der Graphen-Klasse erben und damit die Erzeugung von Objekten ermöglichen, die Kanten zwischen ihnen repräsentieren. Dies ist eigentlich eine sehr elegante Methode um Graphen auf beliebigen Objekten zu definieren.

Bei diesem Vorgehen wird jedoch für jede Kante im Graphen ein einzelnes Objekt erzeugt und gespeichert. Dies könnte bei großen Modellgraphen - im praktischen Einsatz mit JAMES II wurden schon Modelle mit weit mehr als 20.000 einzelnen Elementen simuliert - zu einem ernsten Speicherplatzproblem führen, da meist der gesamte Graph einer Analyse unterzogen wird und somit komplett im Arbeitsspeicher verfügbar sein muss.

Hinzu kommt, dass die Graphen für Modelle und Infrastruktur während der hier vorgestellten initialen Modellpartitionierung einen vollkommen statischen Charakter besitzen: Anzahl und Anordnung der Knoten und Kanten verändern sich nach der Modell- und Infrastrukturanalyse nicht mehr, weshalb das Hauptaugenmerk bei der Entwicklung dieser Klassen auf einer Optimierung des Speicherplatzbedarfs und der Zugriffsgeschwindigkeit beim Lesen lag.

Daher basieren die hier verwendeten Graphen auf einer Darstellung als *Adjazenzlisten*, die folgendermaßen aufgebaut ist: Sei $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$ ein Graph, so ist die Adjazenzliste von G eine Liste $L = (L_1, \ldots, L_n)$, die für jeden im Graphen vorhandenen Knoten v_i eine Liste $L_i = (v_1^i, \ldots, v_k^i)$ mit allen Nachbarknoten von v_i enthält. Näheres zu Adjazenzlisten von Graphen kann in [24] gefunden werden.

Diese einfache Art des Darstellung bietet den großen Vorteil, dass ihr Speicherplatzbedarf nur linear mit der Anzahl von Knoten bzw. Kanten wächst. Dies ist von entscheidendem Vorteil, da bei anderen Datenstrukturen, beispielsweise den zuvor implementierten Adjazenzmatrizen (siehe [24]), ein schnellerer Zugriff auf die Daten durch eine quadratisch wachsende Größe der Datenstruktur erkauft wird. Dies hat sich während des Testens als untauglich für größere Graphen erwiesen.

Außerdem ist es notwendig, dass Knoten mit den Objekten, die sie symbolisieren, assoziiert werden können. Geschieht dies nicht, hat man am Ende zwar einen gewichteten Graphen partitioniert, kann daraus aber keine Schlüsse über die Partitionierung des Modells ziehen, da nicht mehr bekannt ist, welcher Knoten welchen Teil des Modells repräsentiert. Um dies zu realisieren, wird eine Abbildung $Knoten \rightarrow Objekt$ in der `Graph`-Klasse gespeichert.

Knotenbeschriftungen werden in einer einfachen Liste gespeichert. Damit

die Kantenbeschriftungen einfach gehandhabt werden können, wird für jeden Knoten im Graphen eine Abbildung $Nachbar \rightarrow Beschriftung$ gespeichert. Der Datentyp der Beschriftung ist `Double`, so dass ihr Nutzen - wie hier benötigt - auf das Speichern numerischer Beschriftungen beschränkt ist.

Natürlich hätten die Kantenbeschriftungen auch analog zu den Adjazenzlisten gespeichert werden können. Der Nachteil bestünde dann aber in einem komplizierteren und langsameren Zugriff auf die Daten, da die Kanteninformationen nur mit Hilfe der Adjazenzlisten durchsucht werden könnten. Ansonsten wäre unbekannt, welche Beschriftung in der Liste zu welchem Nachbarn gehört.

Ein Nachteil der gewählten Methode ist sicherlich, dass es nur möglich ist, *eine* Beschriftung zwischen zwei Knoten zu speichern. Damit versagt diese Methode bei der Beschriftung von Graphen mit Mehrfachkanten. Da in dieser Arbeit aber nur schlichte Graphen von Belang sind, spielt dies eine untergeordnete Rolle.

Eine Zusammenfassung der Beschriftungsmöglichkeiten, und was diese im Kontext der Partitionierungsalgorithmen bedeuten, ist in Tabelle 4.1 abgebildet.

Art der Beschriftung	**Bedeutung**
Knoten im Infrastrukturgraph	Rechenkapazität eines Prozessors
Knoten im Modellgraph	Berechnungsaufwand eines Modellelements
Kante im Infrastrukturgraph	Qualität der Netzwerkverbindung zwischen Prozessoren
Kante im Modellgraph	Ausmaß der Kommunikation zwischen Modellelementen

Tabelle 4.1: Die Beschriftung jedes Elements hat jeweils im Modell- und Infrastrukturgraph eine unterschiedliche Bedeutung. Die Gewichte sollten von den Algorithmen relativ (also in Beziehung zur Gesamtsumme im Graphen) betrachtet werden, um ihre Anpassungsfähigkeit an verschiedene Umgebungen zu gewährleisten.

Da die Menge der gespeicherten Informationen für einen gewichteten Graph durch Speicherung der Beschriftungen signifikant größer ist als die für einen ungewichteten Graph, wurde zunächst eine allgemeine Klasse `Graph` implementiert, die mit den verschiedenen Eigenschaften wie Einfachheit, der Unterstützung gerichteter Kanten, Kreisfreiheit etc. konfiguriert werden kann.

Von `Graph` wird eine zweite Klasse, `LabeledGraph`, abgeleitet, die diese dann um die beschriebenen Funktionen für die Beschriftung von Kanten und Knoten ergänzt. Diese einfache Klassenstruktur zur Repräsentation von Graphen ist in Abbildung 4.1 verdeutlicht.

Bei Graphen als Eingabedaten für Partitionierungsalgorithmen kommt hinzu, dass man zwar gewisse Eigenschaften, wie Schlichtheit, voraussetzen kann, andere Eigenschaften, die für konkrete Algorithmen auch interessant sind, aber von Fall zu Fall verschieden sein können.

So haben DEVS-Modelle per Definition einen Modellbaum zu Grunde liegen. Bei zellulären Automaten haben die Modellgraphen hingegen eine Gitterstruktur. Solche Merkmale könnten sich als Voraussetzung für die korrekte Funktionsweise eines Partitionierungsalgorithmus herausstellen.

Es ist also wichtig, die verschiedenen Eigenschaften eines Graphen in einem Objekt darstellen zu können, und diese notfalls auch zu kontrollieren. Ein Bei-

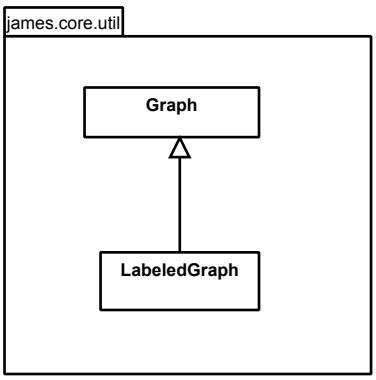

Abbildung 4.1: Die Klassenstruktur zur Repräsentation von Graphen berücksichtigt lediglich Graphen mit Kanten- und Knotenbeschriftung als spezielle Graphenklasse

spiel hierfür wäre die oben erwähnte Kreisfreiheit als notwendiges Merkmal von Bäumen.

Diese Eigenschaft muss gelten, wenn man Tiefensuche oder ähnliche rekursive Vorgehensweisen im Partitionierungsalgorithmus benutzt. Sie kann deshalb für einen Graphen festgelegt werden, so dass der Kreiserkennungsalgorithmus in der Klasse `Graph` automatisch die Korrektheit dieser Eigenschaft sicherstellt.

Es sei noch angemerkt, dass diese Art der Implementierung keineswegs als optimal angesehen werden kann. Sie ist sehr problemspezifisch und schlecht erweiterbar. Da die Repräsentation von Graphen aber für das eigentliche Thema, die Modellpartitionierung, lediglich eine periphere Rolle spielt, wurde an dieser Stelle auf eine aufwendigere und bezüglich möglicher Verwendungszwecke ausgewogenere Entwicklung verzichtet.

4.3 Testverfahren und Darstellung der Ergebnisse

Um die in Abschnitt 3.5 beschriebenen Zusammenhänge testen zu können, mussten zunächst diverse Aufgaben gelöst werden: es wurde eine angemessene Darstellung der Ergebnisse benötigt, Komponenten zur Generierung zufälliger Bäume und Graphen mit gewissen Eigenschaften, sowie eine automatisierte Ausführung der Tests. Außerdem sollten die Testergebnisse automatisch bezüglich den in 3.5 definierten Kriterien (Schnittgröße, Ungleichgewicht und Zeitdauer) evaluiert werden.

Zu diesem Zweck wurden zwei Hilfsklassen, `MassiveTesting` und `MassiveTestingUtility`, entwickelt, wobei die Erstere die Automatisierung und Evaluation der Testläufe und die Letztere die restlichen Aufgaben erfüllt.

`MassiveTesting` kann über diverse Kommandozeilenparameter konfiguriert werden und speichert Ein- und Ausgaben der Testläufe eines Testszenarios in einem angegebenen Verzeichnis. Gleichzeitig werden die Qualitätsmerkmale für die einzelnen Testläufe bestimmt und ebenfalls in eine Datei geschrieben.

`MassiveTesting` kann so konfiguriert werden, dass entweder die Anzahl der Nachbarn pro Knoten (bzw. die Anzahl der Kinder pro Blatt, wenn es sich um einen Baum handelt), die Größe des Modells oder die Anzahl der Constraints schrittweise verändert wird. So erhält man - eine genügend große Anzahl an Testläufen vorausgesetzt - einen guten Eindruck von den Abhängigkeiten der verschiedenen Parameter untereinander. Natürlich kann auch die Anzahl der zu nutzenden Partitionsblöcke eingestellt werden.

`MassiveTestingUtility` enthält nicht nur Funktionen zum Generieren von zufälligen Graphen bzw. Bäumen, sondern auch zur „Darstellung“ der Partitionierungsergebnisse. Der Komplexität der Ergebnisse geschuldet, die ja sozusagen Abbildungen vom Modellgraphen auf den Infrastrukturgraphen sind, muss die Möglichkeit bestehen, diese graphisch darzustellen. Da aber die ästhetische graphische Darstellung von Graphen und Bäumen alles andere als eine triviale Aufgabe ist, wurde für diesen Zweck eine externe Software verwendet.

Graphviz [36] ist eine open-source Visualisierungssoftware für Graphen. Sie besteht aus mehreren Komponenten, die jeweils von der Kommandozeile aus aufrufbar und auf bestimmte Klassen von Graphen oder Darstellungsarten spezialisiert sind. Im Folgenden wurde die Zeichenkomponente `dot` verwendet, da sie Bäume und Graphen gleichermaßen gut zeichnet und einfach zu handhaben ist. `MassiveTestingUtility` bietet daher die Möglichkeit, einen Graph und eine auf ihm definierte Partition in einer `dot`-kompatiblen Datei zu speichern. Dabei werden die Knoten des Modellgraphen - je nach zugewiesenem Partitionsblock - mit unterschiedlichen Farben eingefärbt.

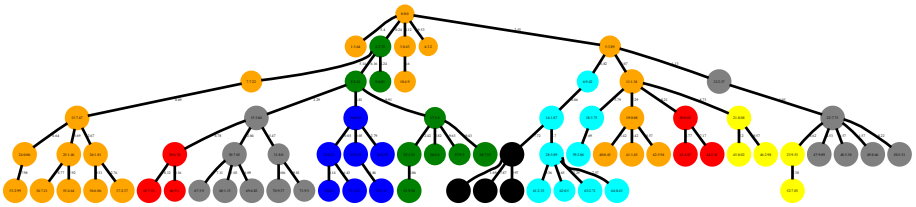

Abbildung 4.2: Beispiel für die Darstellung eines Partitionierungsergebnisses: Die Farben zeigen an, zu welchem Partitionsblock ein Modellelement zugewiesen wurde. Die Zahlen in den Modellelementen bezeichnen die Berechnungskosten, die Zahlen an den Kanten die entsprechenden Kommunikationskosten

Abbildung 4.2 zeigt ein Beispiel für eine auf diese Art generierte Darstellung eines Partitionierungsergebnisses. Wegen des großen Formats der Darstellungen sind diese - zusammen mit den anderen bei Durchführung der Testserien gespeicherten Daten - auf der beiliegenden CD gespeichert.

4.4 Schnittstelle zum Restsystem

Mathematisch gesehen ist eine Partition nichts anderes als eine Abbildung von den Knoten des Modellgraphen auf die Knoten des Infrastrukturgraphen (siehe Abschnitt 3.2.1). Deswegen ist die im `partitionizer` - Paket definierte Klasse `Partition` (siehe Abbildung 3.2) eine direkte Unterklasse von `java.util.HashMap`, welche die grundlegenden Eigenschaften einer Abbildung

besitzt. Sie wird von der Klasse `Partitionizer` als Ergebnis der Partitionierung an die Ausführungsroutinen von JAMES II zurückgegeben.

Jedoch gab es zum Zeitpunkt der Implementierung bereits ein einfaches System zur Partitionierung von Modellen, welches sogar schon in den Ausführungsprozess integriert und getestet worden war. Die nahtlose Integration in die Mechanismen zur Erzeugung der Simulations-Objekte für die verschiedenen Modellierungsformalismen ist recht umfangreich und sensibel, da diese Mechanismen ein zentraler Bestandteil von JAMES II sind und an die Struktur des alten Datentyps für Partitionierungsergebnisse angepasst wurden.

Der bereits vorliegende Datentyp für Partitionen stellt eine Baumstruktur dar und keine Abbildung. Daher werden die neuen `Partition`-Objekte - zumindest bis alle Komponenten in JAMES II mit ihnen umgehen können - nach der Partitionierung in `ExecutablePartition` - Objekte umgewandelt.

Die Klasse `ExecutablePartition` ist eine reine Hilfsklasse, die nur übergangsweise genutzt werden sollte. Sie ist als Unterklasse der alten Klasse für Partitionierungsergebnisse definiert. Dem Konstruktor von `ExecutablePartition` wird das Ergebnis in Form eines `Partition`-Objektes übergeben und dieses wird dann in die gewünschte Struktur umgewandelt.

Somit muss zur Integration des Partitionierungsframeworks in JAMES II lediglich die Variable des alten Partitionierungsobjekts in der Klasse `Simulation` durch eine Instanz von `ExecutablePartition` ersetzt werden, welche dann das mit `Partitionizer.partitionize(...)` erzeugte Ergebnis der Partitionierung in eine für andere JAMES II Komponenten interpretierbare Form umwandelt.

Nachdem diese Vorgehensweise erfolgreich getestet wurde, konnte sich danach auf das Testen des eigentlichen Frameworks beschränkt werden.

Kapitel 5

Ein Algorithmus für DEVS-Modelle

Dieses Kapitel beschreibt einen Algorithmus zur Partitionierung von DEVS-Modellen. Er wurde schrittweise aus neuen Ideen und dem Ansatz von Zeigler et al. ([14], [15]) entwickelt, um ein Gefühl für die generelle Realisierbarkeit der in Abschnitt 3.2.1 formulierten Anforderungen zu entwickeln.

Außerdem sollte neben dem eher simplen und nicht besonders leistungsfähigen Algorithmus zur Partitionierung von Nicht-DEVS-Modellen (siehe Abschnitt 4.1) ein zweiter, spezialisierter Algorithmus ins Framework eingebunden werden. Da für JAMES II oftmals DEVS-Modelle verwendet werden, verspricht ein guter Partitionierungsalgorithmus für diesen Formalismus besonders vorteilhaft zu sein.

Der hier vorgestellte Algorithmus nutzt eine wichtige Eigenschaft von DEVS-Modellen, die Baumstruktur ihrer Modellgraphen, aus. Da diese Einschränkung auf Modellbäume die einzige Eigenschaft ist, die der Algorithmus voraussetzt, kann man ihn prinzipiell zur Partitionierung aller Modelle, die durch einen Modellbaum repräsentiert werden, verwenden.

Das Hauptaugenmerk bei der Entwicklung des Algorithmus lag, entsprechend den erarbeiteten Anforderungen, auf der Vermeidung unnötiger Kommunikation, also der Minimierung der Kosten aller Kanten zwischen den Partitionsblöcken.

Zum besseren Verständnis wurden Flussdiagramme für die einzelnen Fragmente des Algorithmus erstellt. In Anhang B befindet sich eine Zusammenfassung der Diagramme.

5.1 Genereller Ablauf

Der Algorithmus besteht aus drei Phasen und einer Vorstufe, in der die Knoten des Baumes mit Attributen versehen werden, die für die nachfolgenden Phasen von Bedeutung sind. Damit wird die in Abschnitt 3.4.2 beschriebene Möglichkeit der Attributierung von Modellen zur Verfügung gestellt.

Die drei Phasen des Algorithmus werden hintereinander ausgeführt, wobei die letzte und die vorletzte Phase gegebenenfalls mehrmals hintereinander ausgeführt werden müssen:

- Attributierung der Knoten des Baumes (Vorstufe)
- Die Top-down - Phase (Phase 1)
- Die Entscheidungsphase (Phase 2)
- Die Bottom-up - Phase (Phase 3)

Das Grundprinzip des Algorithmus besteht darin, sich zunächst eine bestimmte Ebene des Baumes auszusuchen, die zur Berechnung der Partition verwendet wird. Die gewählte Ebene wird im Folgenden als *Entscheidungsebene* bezeichnet. Die Auswahl geschieht in der *Top-down - Phase*.

Danach wird in der *Entscheidungsphase* darüber entschieden, wie die Knoten der Entscheidungsebene auf die vorhandenen Partitionsblöcke aufgeteilt werden. Alle Knoten, die unterhalb der Entscheidungsebene liegen, werden den Blöcken ihrer Vorfahren in der Entscheidungsebene zugewiesen.

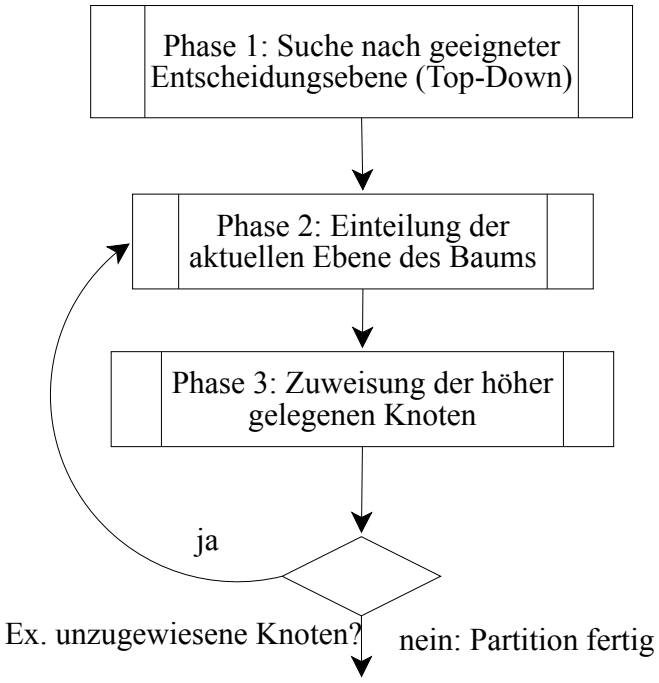

Abbildung 5.1: Allgemeiner Ablauf des Algorithmus

Schließlich werden in der *Bottom-up - Phase* alle Knoten, die sich zwischen der Entscheidungsebene und der Wurzel des Baumes befinden, und daher noch keinem Block zugeordnet wurden, auf die vorhandenen Blöcke aufgeteilt. Dies geschieht durch eine von der Entscheidungsebene ausgehende Analyse der bereits partitionierten Kindknoten.

Dabei kann der Fall auftreten, dass einige Knoten nicht ohne erneute Ausführung der Entscheidungsphase zugewiesen werden können, weshalb diese dann auf den entsprechenden Knoten noch einmal wiederholt wird. Dies wird in Abschnitt 5.4 genauer erläutert.

Das Flussdiagramm in Abbildung 5.1 veranschaulicht den allgemeinen Ablauf. Die Abschnitte 5.3 - 5.5 beschreiben die einzelnen Zwischenschritte und deren Zweck genauer, die Grundidee zur Partitionierung wird in 5.2 erklärt.

5.2 Grundidee zur Partitionierung

Wenn man mit bloßem Auge einen Baum in verschiedene Teile zerlegen will, und dabei so wenig Kanten wie möglich schneiden möchte, sucht man instinktiv nach „passenden Ästen“ innerhalb des Baumes.

Dies ist in diesem Fall tatsächlich eine günstige Vorgehensweise, da die Partitionierung des Modellbaums in möglichst große Teilbäume, wie in 3.2.1 begründet wurde, günstig für die verteilte Simulation von DEVS-Modellen ist. Abbildung 5.2 veranschaulicht die intuitive Herangehensweise.

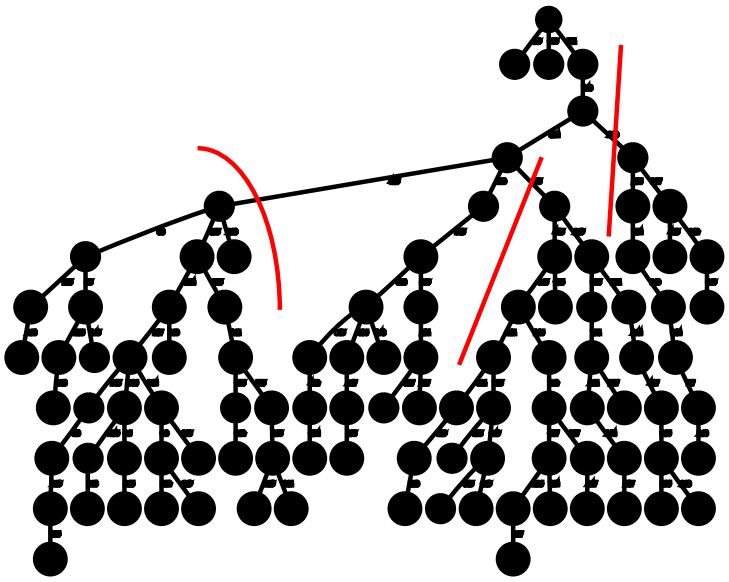

Abbildung 5.2: Eine intuitive Einteilung eines Modellbaums, die Zerlegung in Teilbäume ist gekennzeichnet durch die roten Linien

Um diese Vorgehensweise algorithmisch nachahmen zu können, wird für jeden Knoten die Länge des Weges zu den links und rechts in seiner Ebene angeordneten Knoten berechnet. Die Knoten einer Ebene sind so angeordnet, dass sie die Reihenfolge in der Ebene ihrer Elternknoten widerspiegeln.

Die Anordnung der Kindknoten untereinander ist beliebig. Das heißt, wenn ein Knoten links von einem anderen Knoten der Ebene positioniert ist, dann haben beide entweder den gleichen Elternknoten, oder der des linken Knotens ist in der darüber liegenden Ebene weiter links angeordnet als der des rechten.

Die Anordnung der Knoten in den Ebenen bleibt während des gesamten Algorithmus konstant. Da jede Ebene nun also auch eine spezielle Anordnung der Knoten definiert, werden sie im Folgenden als *Tupel* interpretiert und dementsprechend verwendet.

Es gibt in einem Baum $B = (V, E)$ genau einen Weg zwischen zwei Knoten $v_i, v_j \in V$. Dessen Länge, $l(v_i, v_j)$, lässt sich leicht berechnen: Bezeichne L das Tupel der Knoten einer Ebene, $L = (v_1, \ldots, v_k)$, mit $v_1, \ldots, v_k \in V$. Außerdem sei die Funktion $p : V \to V$ definiert:

$$p(v) = \begin{cases} \text{Elternknoten von v, wenn v nicht die Wurzel des Baumes ist} \\ v \text{ sonst} \end{cases},$$

Mit diesen Mitteln ist die Weglänge zwischen zwei Knoten $v_i, v_j \in L$, wobei o.B.d.A. $i < j$ gilt, wie folgt berechnen:

$$l(v_i, v_j) = \begin{cases} 0, \text{wenn } v_i = v_j \\ 2, \text{wenn } p(v_i) = p(v_j) \\ l(p(v_i), p(v_j)) + 2, \text{ sonst} \end{cases}$$

Wie man sieht, lässt sich die Weglänge sehr leicht rekursiv berechnen, dies wird in Abschnitt 5.3 genauer erklärt. Die Weglänge zwischen zwei Knoten einer Ebene wird im Folgenden als die *Distanz* der beiden Knoten zueinander bezeichnet.

Betrachtet man nun eine einzelne Ebene aus der „Mitte“ des Baumes in Abbildung 5.2, beispielsweise die Dritte von unten, lässt sich die in der Skizze rot angedeutete Zerlegung des Baumes an den größten Distanzen zwischen den Knoten dieser Ebene ablesen.

Dieser Sachverhalt ist nicht weiter verwunderlich, schließlich haben zwei Knoten einer Ebene, deren Distanz d beträgt, erst $\frac{d}{2}$ Ebenen höher einen gemeinsamen Vorfahren (die Weglänge zweier Knoten einer Ebene ist immer durch 2 teilbar). Ist d besonders groß, reichen die „Äste“ der Knoten also bis weit nach oben, ohne aufeinander zu treffen. Dies ist genau das Kriterium, dass in Abschnitt 3.2.1 für die Gestalt der Partitionsblöcke für DEVS - Modelle gefordert wurde.

Diese Idee wird in der Entscheidungsphase genutzt, um die Knoten einer Ebene auf verschiedene Partitionsblöcke aufzuteilen. Durch die Nutzung der Distanzen ist es allerdings notwendig, dass der Algorithmus den Baum Ebene für Ebene analysiert - andernfalls wären die Distanzen wenig aussagekräftig. Neben den verschiedenen Heuristiken zur Erzeugung der Partition ist dies der größte Unterschied zum Verfahren von Zeigler et al., wo der Baum nur entlang bestimmter Pfade weiter analysiert wird.

5.3 Top-down - Phase

Die Funktion der Top-down - Phase leitet sich aus der Tatsache ab, dass sich der Modellbaum am günstigsten von der Wurzel ausgehend analysieren lässt. Das liegt daran, dass die Zahl der analysierten Knoten minimiert werden soll, so dass die Laufzeit des Algorithmus nicht zu stark von der Größe des Modellbaums abhängt. Durch die Analyse von oben nach unten kann der Algorithmus sicherstellen, dass nicht mehr Knoten zur Partitionierung betrachtet werden müssen als nötig.

Die Berechnung der Distanzen (siehe Abschnitt 5.2) zwischen Knoten einer Ebene lässt sich aufgrund ihrer rekursiven Natur sehr einfach in der Top-down-Phase implementieren, so dass der Algorithmus in diesem Schritt die Entfernung

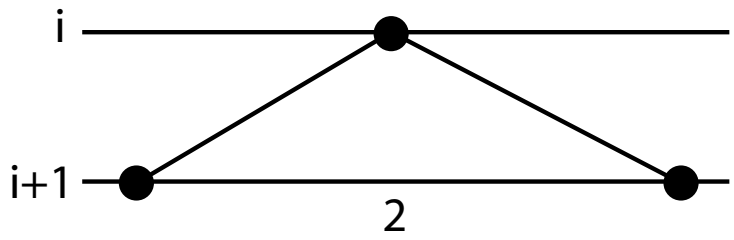

Abbildung 5.3: Die Distanz zwischen zwei Knoten mit dem gleichen Elternknoten ist zwei

zwischen den Knoten einer Ebene schnell ermitteln kann. Die Abbildungen 5.3 und 5.4 veranschaulichen die Berechnung.

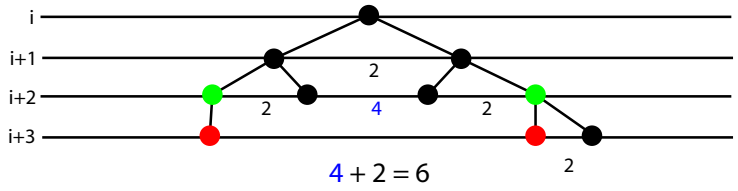

Abbildung 5.4: Die Distanz zwischen zwei Knoten (rot) mit verschiedenen Elternknoten (grün) ergibt sich aus der Distanz zwischen den Elternknoten, die um zwei erhöht wird

Ziel der Top-down - Phase ist es, eine günstige Entscheidungsebene für die Entscheidungsphase zu finden. Es wird mit dem Wurzelknoten begonnen, dann wird der Baum ebenenweise analysiert. Dies geschieht solange, bis eine Ebene gefunden wird, welche die Merkmale einer Entscheidungsebene erfüllt. Diese Merkmale werden nun genauer erläutert.

Das wichtigste Kriterium für die Wahl der Entscheidungsebene wird durch die Beachtung von Constraints notwendig. Im Folgenden sind mit Constraints *konkrete* Constraints gemeint. Unbestimmte Constraints werden durch den Algorithmus vor Beginn der Berechnungen in konkrete Constraints umgewandelt, welche die betroffenen Knoten zu den jeweils am besten passenden Partitionsblöcken zuweisen. Diese Vorgehensweise stellt einen Kompromiss dar: Sie erlaubt dem Nutzer zwar die Eingabe von unbestimmten Constraints, erhöht jedoch nicht die Komplexität des Algorithmus.

Um für die Entscheidungsphase nutzbar zu sein, dürfen in den Teilbäumen *unterhalb* der in Frage kommenden Ebene keine *Konflikte* existieren. Ein Konflikt in einem Teilbaum ist dann gegeben, wenn seine mit Constraints festgelegten Knoten *verschiedenen* Blöcken zugewiesen sind.

Der Ausdruck Konflikt bezieht sich daher auf die Tatsache, dass der Teilbaum nicht komplett einem Block zugewiesen werden kann, ohne die definierten Constraints zu verletzen. Sollten sich im Teilbaum jedoch nur Knoten befinden, die über Constraints dem selben Block zugewiesen wurden, liegt *kein* Konflikt

vor - statt dessen wird der Knoten auf der Entscheidungsebene als bereits diesem Block zugewiesen behandelt. Eine Veranschaulichung dieses Sachverhalts ist in Abbildung 5.5 zu sehen.

Diese wichtige Einschränkung ist dadurch zu erklären, dass *alle* unterhalb der Entscheidungsebene angeordneten Knoten mit der Partitionierung der Entscheidungsebene implizit dem Block ihres Vorfahren auf der Entscheidungsebene zugewiesen werden sollen. Dies kann aber nur geschehen, wenn unterhalb der Entscheidungsebene keine Konflikte zwischen bereits zugewiesenen Knoten existieren.

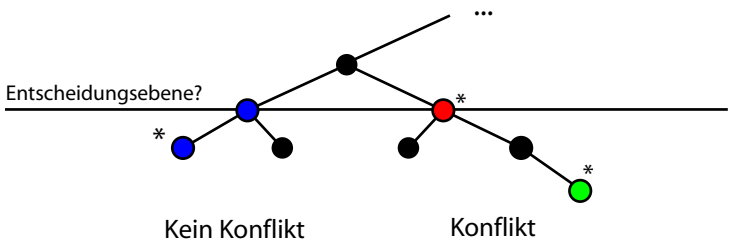

Abbildung 5.5: Beispiel für Constraint-Konflikt in Top-down Phase: Die mit * markierten Knoten wurden mittels Constraints fixiert. Links sieht man, dass das Vorhandensein eines Constraints unterhalb der Entscheidungsebene kein Problem ist - das Elternteil und damit der gesamte Teilbaum werden diesem Block zugewiesen. Rechts dagegen widersprechen sich die Constraints, es gibt einen Konflikt - die Entscheidungsebene muss hier tiefer gesucht werden, um den Konflikt so gut wie möglich zu lösen.

Die Top-down - Phase kann also erst beendet werden, wenn die Constraints für unter der aktuellen Ebene liegende Knoten keine Probleme verursachen. Ist dies der Fall, wird zusätzlich noch die Größe der Ebene, also die Anzahl der in ihr enthaltenen Knoten, beachtet. Dieses Kriterium ist allerdings nicht entscheidend; es wird nur angewandt, wenn bereits eine konfliktfreie Ebene erreicht wurde.

Idealerweise sollte eine Ebene gerade groß genug sein, damit sie gut auf die vorhandenen Partitionsblöcke aufgeteilt werden kann. Doch wann genau ist eine Ebene „groß genug“? Das ist für den allgemeinen Fall schwer zu sagen und stark von der Beschaffenheit des Modells abhängig. Ist das Modell sehr gleichmäßig aufgebaut, das heißt die Blätter des Modellbaums haben fast alle die gleiche Tiefe und die Kosten der einzelnen Knoten variieren nur minimal, können schon sehr wenige Knoten für eine gute Partitionierung ausreichen.

Dies muss aber nicht der Fall sein, daher wird der Parameter $idealNum_{PB}$ (siehe Tabelle 5.1 auf Seite 78) eingeführt, der das gewünschte Verhältnis zwischen der Größe der Entscheidungsebene und der Anzahl der Partitionsblöcke bestimmt. Der Baum wird nur so lange herabgestiegen, bis eine konfliktfreie Ebene gefunden wird, die als groß genug zur Ausführung der Entscheidungsphase betrachtet werden kann.

Falls aber eine Ebene - bevor dieses Kriterium erfüllt werden konnte - weniger Knoten als die vorige besitzt, wird die vorige Ebene, sozusagen als lokales Maximum der Knotenzahl, als Entscheidungsebene genutzt. Dies ist dadurch zu

motivieren, dass ein zu tiefes Herabsteigen, und damit eine „unrepräsentative“ Ebene wie in Abbildung 5.6, nach Möglichkeit vermieden werden sollte.

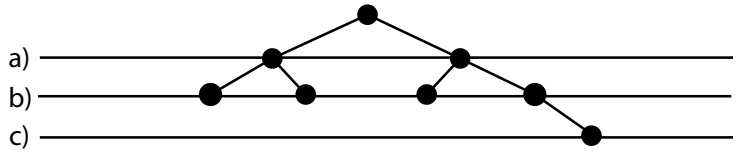

Abbildung 5.6: Wird a) als Entscheidungsebene gewählt, sind zu wenig Knoten vorhanden. Wird wie in c) zu tief herabgestiegen, ist die Entscheidungsebene nicht mehr aussagekräftig. In diesem Baum wäre b) die beste Entscheidungsebene.

Wann eine für die Entscheidungsphase günstige Ebene gefunden wird, hängt also alles in allem von $idealNum_{PB}$, der Struktur des Modellbaums, den definierten Constraints und der Anzahl der zu verwendenden Partitionsblöcke ab.

Trotz dieses komplexen Kriteriums ist es leider leicht möglich, Situationen zu finden, in denen die vorgeschlagene Herangehensweise nicht gut funktioniert (siehe Abbildung 5.7).

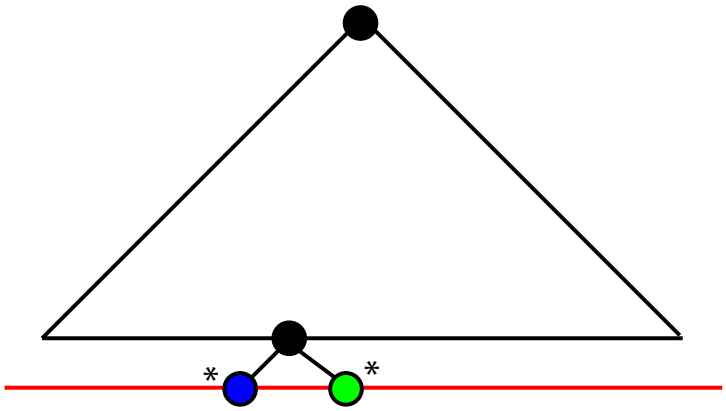

Abbildung 5.7: Schwieriger Fall für die Wahl der Entscheidungsebene: Da die untersten beiden Knoten verschiedenen Blöcken zugewiesen wurden, muss eine Entscheidungsebene (rote Linie) unterhalb des restlichen Baumes (skizziert durch das Dreieck) gewählt werden.

Die Vermeidung dieser Sonderfälle kann durch die Hinzunahme weiterer Kriterien verhindert werden. Beispielsweise sind besonders ungünstige Konflikte (wie in Abbildung 5.7) anhand der Tiefe der Knoten, die sie verursachen, zu erkennen. Sie könnten zunächst außen vor gelassen werden, und ihre Teilbäume wären dann später separat zu partitionieren.

Damit der Algorithmus übersichtlich bleibt, wurden statt dessen (siehe Abschnitt 5.4 und 5.5) seine Partitionierungsmethoden so verändert, dass diese

und ähnliche Situationen weniger gravierende Folgen haben.

5.4 Entscheidungsphase

Dies ist die schwierigste und aufwändigste Phase des Algorithmus, da hier die generelle Zuordnung der Knoten zu den Partitionsblöcken festgelegt wird. Dazu werden folgende Informationen genutzt:

- Rechenkapazität der Prozessoren sowie die Kapazität ihrer Kommunikationsverbindungen
- Berechnungsaufwand der Modelle
- Constraints

Die Entscheidungsphase selbst wird in drei Runden eingeteilt. Auch hier spielen die definierten Constraints eine zentrale Rolle.

Knoten auf der Entscheidungsebene, die durch Constraints zugeordnet wurden, besitzen dabei die höchste Priorität: Ihren Blöcken sollten möglichst viele bezüglich der Entscheidungsebene benachbarte Knoten zugewiesen werden. Das Auffüllen der Blöcke, die ihnen durch Constraints zugewiesene Knoten auf der Entscheidungsebene besitzen, geschieht in Runde eins. Danach werden diese Blöcke als `fertig` markiert, das heißt sie werden von den weiteren Runden als bereits bearbeitet betrachtet.

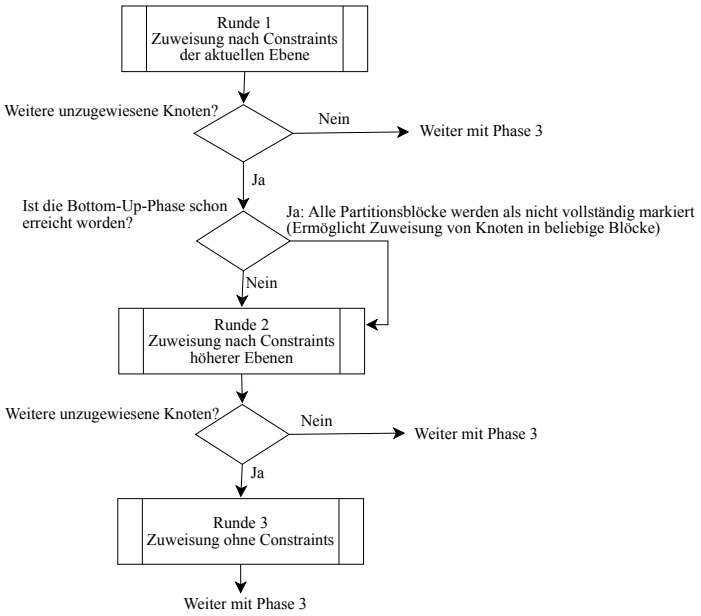

Abbildung 5.8: Flussdiagramm für die Entscheidungsphase

Die zweite Runde versucht, die Constraints für Knoten aus höher gelegenen Ebenen zu berücksichtigen, um die übrig gebliebenen Knoten möglichst „vorausschauend“ den übrig gebliebenen Blöcken zuzuweisen.

Die dritte und letzte Runde identifiziert schließlich die restlichen zuzuordnenden Knoten und weist diese den jeweils günstigsten Blöcken zu.

Diese Abfolge der Abarbeitung ist im Flussdiagramm in Abbildung 5.8 verdeutlicht.

5.4.1 Runde 1: Ausweitung der Blöcke mit bereits zugeordneten Knoten

Zunächst wird eine Liste aller bereits zugeordneter Knoten der Entscheidungsebene angelegt. Ein Knoten ist auch dann zugeordnet, wenn einer der Knoten im unter ihm gelegenen Teilbaum einem Block zugewiesen ist, damit keine Konflikte wie in Abschnitt 5.3 beschrieben auftreten.

Danach werden alle Blöcke ermittelt, zu denen die Knoten der erstellten Liste zugeordnet werden. Diese Blöcke werden innerhalb dieser Runde mit weiteren Knoten aufgefüllt und dann als `fertig` markiert.

Es wird daher versucht, jeden Block so weit wie möglich zu vergrößern, indem ihm bislang nicht zugeordnete Knoten zugewiesen werden.

Um abzuschätzen, wie viele Knoten einem einzelnen Block b zugeordnet werden sollten, wird sein Anteil an der Gesamtrechenkapazität ($\sigma_{cap,calc}$) auf die Gesamtberechnungskosten der Knoten in der Entscheidungsebene projiziert. Daraus ergibt sich dann eine Obergrenze ($limit$) für die gesamten Berechnungskosten des Blocks bezüglich der Entscheidungsebene (alle verwendeten Symbole sind wie in 3.2.1 definiert):

$$
\begin{aligned}
&v_b^I \in V^I - \text{den Block b repräsentierender Knoten}\\
&L - \text{in Phase 1 gewählte Entscheidungsebene}\\
&\sigma_{cap,calc} = \textstyle\sum_{v_i^I \in V^I} cap_{calc}(v_i^I)\\
&\sigma_{cost,calc}(L) = \textstyle\sum_{v_i^M \in L} cost_{calc}(v_i^M)\\
&limit = \sigma_{cost,calc}(L) \cdot \frac{cap_{calc}(v_b^I)}{\sigma_{cap,calc}}
\end{aligned}
$$

Es handelt sich also um eine einfache Interpolation. Nun wird versucht, dem Partitionsblock so viele Knoten hinzuzufügen, bis die Summe des Berechnungsaufwands der ihm zugewiesenen Knoten größer als $limit$ ist.

Es werden jedoch nicht wahllos Knoten hinzugefügt. Betrachtet werden nur die Knoten, die mit mindestens einem Knoten aus dem Block bezüglich der Entscheidungsebene benachbart sind.

Existiert kein Knoten in der Entscheidungsebene, der höchstens das $maxDist_{nextNode}$-fache (siehe Tabelle 5.1 auf Seite 78) des durchschnittlichen Abstands auf der Entscheidungsebene entfernt ist, wird abgebrochen. Ansonsten wird der Knoten mit dem geringsten Abstand dem Block zugewiesen. Gibt es mehrere Knoten mit gleichem Abstand, wird aus ihnen der Knoten mit dem geringsten Berechnungsaufwand gewählt.

Außerdem wird darauf geachtet, dass keiner der dem Block hinzugefügten Knoten näher an einem ebenfalls durch Constraints fixierten Knoten eines anderen Blockes liegt. Dies führt dazu, dass die Reihenfolge der Blöcke beim Auffüllen keine so starke Rolle spielt, weil viele der in Frage kommenden Knoten bereits für den entsprechenden Block „reserviert“ werden.

Insgesamt wird durch diese Regeln versucht, günstige unzugeordnete Knoten zusammen mit den durch Constraints festgelegten Knoten zu Blöcken zu aggregieren, die möglichst große Teilbäume des Baumes beinhalten. Der Parameter

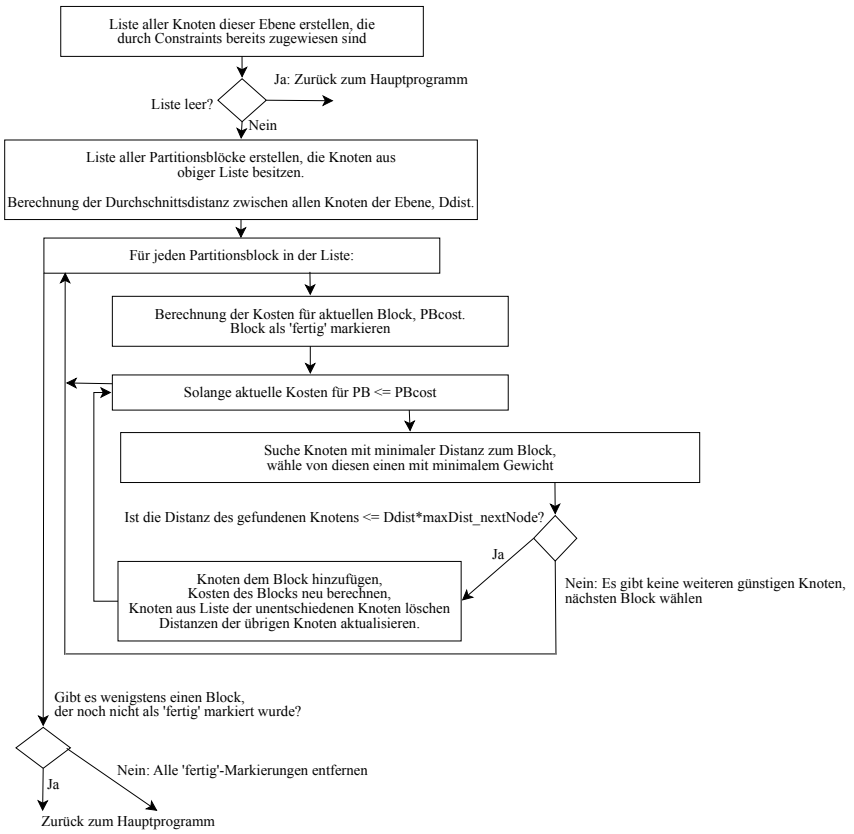

Abbildung 5.9: Flussdiagramm für die 1. Runde der Entscheidungsphase

$maxDist_{nextNode}$ beeinflusst die Anzahl der Knoten, die einem Block hinzugefügt werden können, da er die maximale Entfernung, die ein hinzuzufügender Knoten besitzen darf, in Relation zum Durchschnittsabstand festlegt.

Am Ende der ersten Runde sind die Blöcke, die durch Constraints zugewiesenen Knoten enthalten, zunächst vollständig bearbeitet: Sie werden als `fertig` markiert. Dies ist jedoch keine endgültige Markierung, sondern sie kann unter gewissen Bedingungen (siehe Abschnitt 5.5) wieder aufgehoben werden.

Das Flussdiagramm in Abbildung 5.9 illustriert den genauen Ablauf der ersten Runde.

5.4.2 Runde 2: Wahl der Prozessoren gemäß Beschränkungen in den höheren Ebenen

Die verbleibenden p' Blöcke, die keine bereits zugewiesenen Knoten enthalten, könnten jetzt ohne Verletzung der Beschränkungen die restlichen Knoten aufnehmen. Allerdings ist es möglich, dass es dabei zu unvorteilhaften Situationen, wie beispielhaft in Abbildung 5.10 skizziert ist, kommen kann.

Es ist also notwendig, auch die Beschränkungen oberhalb der Entscheidungsebene zu beachten. Da während der nachfolgenden Bottom-up - Phase die Zu-

ordnung der höher gelegenen Knoten erfolgt, kann eine vorausschauende Zuordnung dafür sorgen, dass Situationen wie in Abbildung 5.10 weitestgehend vermieden werden.

Für eine intelligentere Verteilung auf die Blöcke wird hier ein Verfahren gewählt, bei dem die Knoten der Entscheidungsebene - falls sie Vorfahren mit fester Zuordnung haben - darüber „abstimmen“, welchem Block sie zugewiesen werden sollen.

Außerdem kann bei der Auswahl eines passenden Blocks auch die Nachbarschaft der bereits zugewiesenen Knoten eine Rolle spielen, da diese weiter oben im Baum gemeinsame Vorfahren haben.

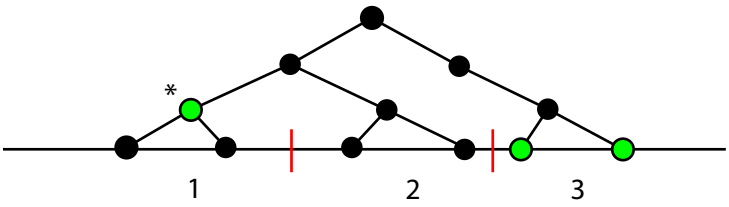

Abbildung 5.10: In diesem Beispiel ist der Knoten oberhalb der Entscheidungsebene durch Constraints dem grünen Block zugeordnet. Dies wird durch ein ’*’ angedeutet. Dem Block könnten jedoch ohne zwingenden Grund Knoten auf der rechten Seite der Entscheidungsebene zugewiesen werden, die weit vom anderen Knoten des Blocks entfernt sind. Dadurch wurde die Chance auf eine minimale Schnittgröße drastisch verringert. Dies soll durch die zweite Runde verhindert werden

Doch erst einmal kommt nun die wichtigste Heuristik des Algorithmus zum tragen, die Analyse der in der Top-down - Phase berechneten Distanzen zwischen den einzelnen Knoten der Entscheidungsebene. Weil durch die bereits in Runde 1 zugeordneten Knoten „Lücken“ in der Entscheidungsebene entstanden sind, werden die Distanzen zwischen den noch zuzuordnenden Knoten neu berechnet.

Aus den aktualisierten Distanzen werden dann die Stellen im Tupel ermittelt, an denen die $p' - 1$ größten Distanzen existieren. Diese Stellen werden im Folgenden auch als die *Zerlegungspunkte* der Entscheidungsebene bezeichnet. Bei gleich großen Distanzen werden die Zerlegungspunkte bevorzugt, welche die Entscheidungsebene „gleichmäßiger“ aufteilen, also einen größeren Abstand zu den anderen Zerlegungspunkten der Ebene besitzen. Die Zerlegung der Entscheidungsebene ist in Abbildung 5.11 veranschaulicht.

Des Weiteren muss auch darauf geachtet werden, dass die Ebene nicht an Punkten mit zu kleiner Distanz zerlegt wird. Daher werden nur Zerlegungspunkte gewählt, bei denen eine gewisse Minimaldistanz zwischen den Knoten existiert.

Die Minimaldistanz wird ermittelt, indem während der Top-down - Phase die Zerlegungspunkte für die Aufteilung jeder analysierten Ebene berechnet werden. Jede Ebene wird hypothetisch für die Gesamtzahl der vorhandenen Blöcke aufgeteilt, und die kleinste Distanz an einem Zerlegungspunkt wird für jede Ebene gespeichert. Eine Ebene, bei der diese *minimale Zerlegungsdistanz* besonders groß ist, eignet sich besonders gut zur Zerlegung, weil die gewählten Teilbäume

erst sehr weit oben im Baum zusammentreffen würden. Als Minimaldistanz wird deshalb die größte minimale Zerlegungsdistanz der Ebenen oberhalb der Entscheidungsebene festgelegt. Die Minimaldistanz wird auch mit $splitDistance_{min}$ bezeichnet.

Gibt es nicht genügend Punkte, die das Minimaldistanz-Kriterium erfüllen, werden weniger Zerlegungspunkte zurückgegeben als angefordert. Dementsprechend wird die Entscheidungsebene auf weniger Blöcke aufgeteilt.

Bildlich gesprochen prüft der Algorithmus dadurch, ob oberhalb der Entscheidungsebene noch bessere Gelegenheiten zur Partitionierung existieren. Auf diese Weise kann die zu starke Partitionierung einer ungünstigen Entscheidungsebene verhindert werden.

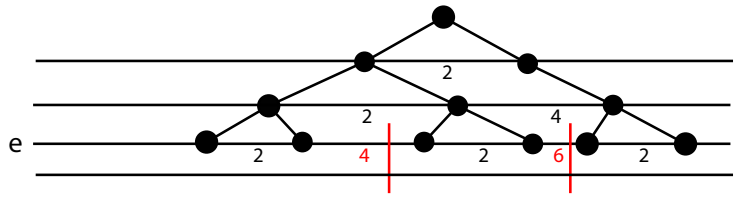

Abbildung 5.11: Die Stellen mit den größten Distanzen zwischen den Knoten werden zur Partitionierung genutzt (bereits zugeordnete Knoten werden in diesem Beispiel nicht dargestellt).

Die Zerlegungspunkte teilen die unzugeordneten Knoten der Entscheidungsebene in *Zonen* ein. Jetzt muss jeder Zone ein möglichst günstiger Block zugewiesen werden. Alle Knoten einer Zone, so wie deren benachbarte Knoten, die bereits einem Block zugeordnet sind, entscheiden über den am besten geeigneten Block. Die Einteilung der gesamten Ebene in „Wahlbereiche“ wird wie in Abbildung 5.12 angedeutet vorgenommen.

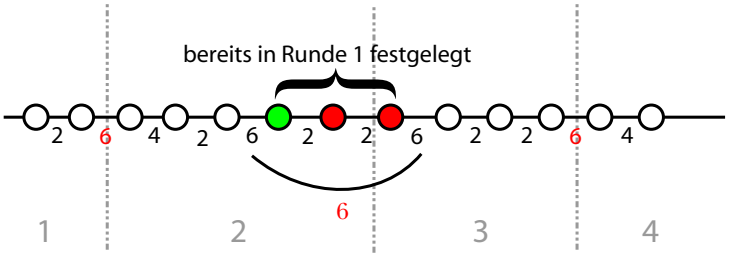

Abbildung 5.12: Hier soll der Graph in 6 Partitionsblöcke zerlegt werden, von denen zwei bereits in Runde 1 festgelegt wurden. Dadurch verändern sich die Distanzen zwischen den übrig gebliebenen Knoten links und rechts davon. Die rot markierten Stellen werden als Zerlegungspunkte zur Aufteilung in die $p' = 4$ verbleibenden Blöcke genutzt. Die grauen Linien markieren die Wahlbereiche.

Ein Knoten darf für den Block jeder seiner durch Constraints zugeordneten Vorfahren eine Stimme abgeben, sofern dieser noch nicht als `fertig` mar-

kiert wurde. Die Stimmen sind nach der Tiefe des Vorfahren gewichtet. Die Beschränkungen tieferer Knoten zu erfüllen hat Vorrang, da diese näher an der Entscheidungsebene liegen. Außerdem werden nur Constraints beachtet, welche die Vorfahren der in der Zone befindlichen Knoten direkt betreffen (siehe Abbildung 5.13).

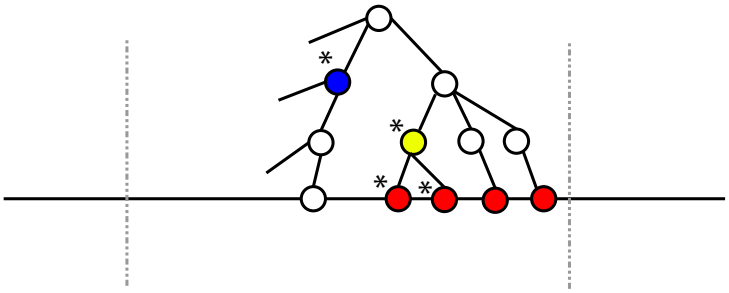

Abbildung 5.13: In Runde 2 der Entscheidungsphase beachtete Constraints: Die grauen Linien markieren die Wahlbereiche, die schwarze Linie die Entscheidungsebene, * kennzeichnet einen Knoten als durch Constraints fixiert. Die rot markierten Knoten wurden bereits in der ersten Runde zugewiesen. Während der für den oberen Knoten festgelegte Block (blau) bei der Wahl eine Rolle spielt, wird die Zuordnung des Knoten zum gelben Block bei der Wahl nicht beachtet. Die Constraints werden also nur beachtet, wenn diese auf Vorfahren von unzugeordneten Knoten der Entscheidungsebene bezogen sind.

Nachdem für jeden Wahlbereich abgestimmt wurde, wird das Wahlergebnis mit der höchsten Stimmzahl umgesetzt, das heißt dem Block, der die Wahl einer Zone mit den meisten Stimmen gewonnen hat, werden alle Knoten der Zone zugewiesen und er wird als `fertig` markiert.

Dies geschieht so lange, bis kein Wahlergebnis mehr umgesetzt werden kann, weil der Wahlsieger bereits einen anderen Abschnitt erhalten hat oder es kein weiteres Wahlergebnis gibt. Die restlichen Knoten werden in der 3. Runde zugeordnet.

Leider gibt es noch einen Sonderfall, der die hier beschriebene Vorgehensweise noch komplexer gestaltet: Angenommen, die Constraints für ein bestimmtes Modell sind so ungünstig wie in Abbildung 5.7. Dann wird die Entscheidungsphase für eine extrem ungünstige Ebene aufgerufen, was auch bedeutet, dass der Algorithmus für die Bottom-Up - Phase kaum Informationen erhält, die für die Partitionierung der höher gelegenen Knoten genutzt werden kann. Um dem Abhilfe zu schaffen, kann während der Bottom-up - Phase auch auf die Algorithmen der Entscheidungsebene zurückgegriffen werden.

Diese müssen jedoch ihr Verhalten an die jeweilige Situation anpassen. Runde 1 und 3 sind allgemein genug, um in jeder Situation verwendet werden zu können. Im der zweiten Runde muss jedoch das Wahlsystem an die neuen Gegebenheiten angepasst werden.

Die Bottom-up - Phase nutzt die Informationen zu den Blöcken der Kindknoten, um Knoten der höheren Ebenen günstig zu partitionieren. Falls während der Bottom-up - Phase auf Knoten gestoßen wird, die keine Kinder besitzen,

wird die Entscheidungsphase für diese Knoten aufgerufen. Diese partitioniert also zusätzlich zur Entscheidungsebene auch alle höher gelegenen *Blätter* des Modellbaums.

Die weiter oben beschriebene Vorgehensweise zum Auffinden geeigneter Zerlegungspunkte ermöglicht es dem Algorithmus, auch diese Ebenen mit Blättern in Zonen zu zerlegen. Dabei stellt sich die Frage, wann sich der Algorithmus dafür entscheiden soll, die ermittelten Zonen neuen Blöcken zuzuweisen, oder dabei vor allem die bereits zugewiesenen Nachbarknoten zu betrachten. Abbildung 5.14 verdeutlicht diese Fragestellung.

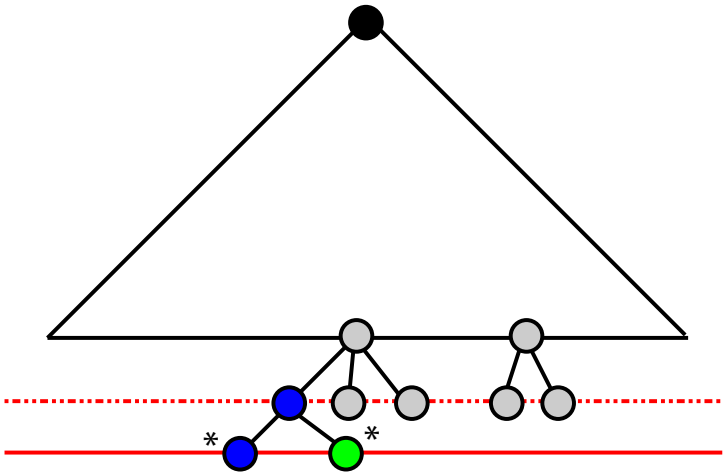

Abbildung 5.14: Entscheidung über Zuweisung von Knoten in neuen Block: Die grauen Knoten sind noch keinem Block zugewiesen. Weil die ursprüngliche Entscheidungsebene (rote ununterbrochene Linie) sehr ungünstig gewählt werden musste, sollte auf den höheren Ebenen (rote gestrichelte Linie) entschieden werden können, in welcher Situation neue Blöcke für die oberhalb liegenden Blätter genutzt werden.

Dies ist eine schwierige Frage, da es sicherlich für die Minimierung der Kommunikation von Vorteil ist, wenn die Blätter kleinerer Teilbäume dem gleichen Block zugewiesen werden wie ihre Nachbarn. Andererseits sollten möglichst alle Blöcke zur Partitionierung verwendet werden.

Für die Entscheidung dieser Frage wird wiederum auf die größte Minimaldistanz, $splitDistance_{min}$, sowie auf die Parameter $ratio_{undecided}$ und $minDist_{newPB}$ zurückgegriffen. Es wird überprüft, ob die Zone genügend unentschiedene Knoten im Verhältnis zu den bereits in der Bottom-up - Phase entschiedenen Knoten besitzt, und ob ihr Teilbaum weit genug nach oben reicht. Näheres zu diesem Kriterium ist der Tabelle 5.1 (Seite 78) zu entnehmen.

Unter einem „neuen Block“ wird im Folgenden ein Block verstanden, der nicht als `fertig` markiert wurde. Weiterhin wird zwischen Zonen unterschieden, die einem neuen Block zugewiesen werden sollen, und Zonen, für die dies nicht gilt. Erstere werden dadurch charakterisiert, dass sie eine große Anzahl unzugewiesener Knoten enthalten. Dies rechtfertigt die Zuweisung eines neuen Blocks, da zwar der Kommunikationsaufwand erhöht wird, aber das Ungleich-

gewicht zwischen den Blöcken verringert werden kann.

Die Art der Zone bestimmt das Verfahren, das zur Wahl eines geeigneten Blocks verwendet wird. Für Zonen, die einem neuen Block zugewiesen werden sollen, wird das bereits oben beschriebene Wahlsystem eingesetzt. Dieses lässt zum Beispiel nur Stimmen für die Constraints direkter Vorfahren zu (siehe Abbildung 5.13) und verhindert die Wahl von Blöcken, die bereits als `fertig` markiert wurden. Das Ergebnis dieser Einschränkungen ist, dass die bereits zugewiesenen Knoten der Zone keinen zusätzlichen Einfluss auf die Wahl eines geeigneten Blocks nehmen können, und dass die Zone einem neuen Block zugewiesen wird.

Im Gegensatz dazu wird das Wahlsystem für eine Zone, die keinem neuen Block zugewiesen werden soll, leicht abgeändert (siehe unten). Ihre unzugewiesenen Knoten werden mit großer Wahrscheinlichkeit einem Block zugewiesen, der bereits Knoten aus der Zone enthält. Die bereits zugewiesenen Knoten der Zone spielen hier also eine maßgebliche Rolle. Abbildung 5.15 verdeutlicht den Unterschied der verschiedenen Situationen.

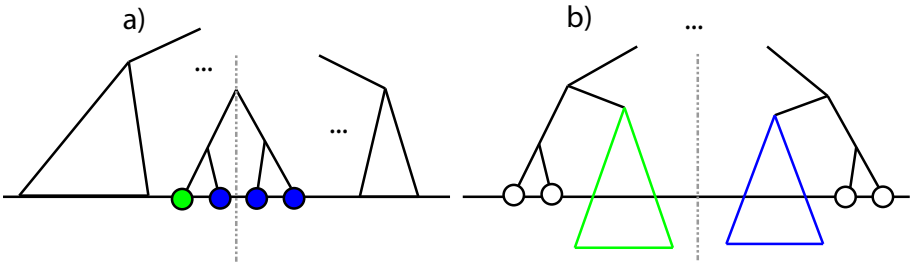

Abbildung 5.15: Fallunterscheidung für die Blockwahl in Zonen: Die aktuell betrachtete Ebene ist jeweils als schwarze, horizontale Linie dargestellt, die Grenze der Zonen als graue, vertikale Linie. Die Dreiecke sollen größere Teilbäume unzugewiesener Knoten (schwarze Linien, Fall a) bzw. zugewiesener Knoten (grüne bzw. blaue Linien, Fall b) andeuten; die Knoten der farbigen Teilbäume sind dabei genau genommen nur bis zur aktuellen Ebene zugewiesen. In Fall a) wäre es günstig, den unzugewiesenen Teilbäumen neue Blöcke zuzuweisen, da diese groß sind. In Fall b) dagegen gibt es wenige unzugeordnete Knoten, die am Besten auf den grünen und den blauen Block aufgeteilt würden. Es ist zu beachten, dass dieser Sachverhalt *nicht* durch die Betrachtung der Distanz zwischen den unzugeordneten Knoten der beiden Zonen auflösbar ist, da diese in beiden Fällen gleich sein kann.

Während der Entscheidungsphase wird *jeder* Zone ein neuer Block zugewiesen, daher gelten die folgenden Absätze nur für das Verhalten der 2. Runde, wenn bereits die Bottom-up - Phase erreicht wurde.

Soll für eine Zone kein neuer Block genutzt werden, sind Situationen wie in Abbildung 5.13 anders zu bewerten: Nun soll der Block ausgewählt werden, der am günstigsten für die benachbarten, bereits zugewiesenen Knoten ist.

Um das Wahlsystem möglichst ähnlich zu gestalten, werden für alle durch die Bottom-up - Phase zugewiesenen Knoten entsprechende Constraints defi-

niert. Ignoriert man nun während der Wahl die durch Abbildung 5.13 illustrierte Eingrenzung der Stimmen, können auch Stimmen für alle in der Bottom-up - Phase definierten Constraints der Nachbarknoten mitgezählt werden. Somit entscheidet die Nachbarschaft einer Zone über die Wahl des am besten geeigneten Blocks.

Eine Voraussetzung dafür ist die Zulassung von Stimmen für bereits gefüllte Blöcke in diesem Abstimmungsmodus. Allerdings werden letztendlich nur die Constraints in Erwägung gezogen, die am dichtesten an den Knoten der Zone liegen.

Insgesamt kann die Funktion der zweiten Runde wie folgt zusammengefasst werden: Eine Ebene des Baumes wird in Zonen aufgeteilt. In der Entscheidungsphase werden die Zonen neuen Blöcken zugewiesen. Während der Bottom-up Phase werden die Zonen entweder analog zur Entscheidungsphase neuen Blöcken zugewiesen (die dann als `fertig` markiert werden), oder sie werden dem Block zugewiesen, der unter Betrachtung der Nachbarschaft der Zone am günstigsten ist. Weitere Details zum Ablauf des Algorithmus lassen sich Abbildung 5.16 entnehmen.

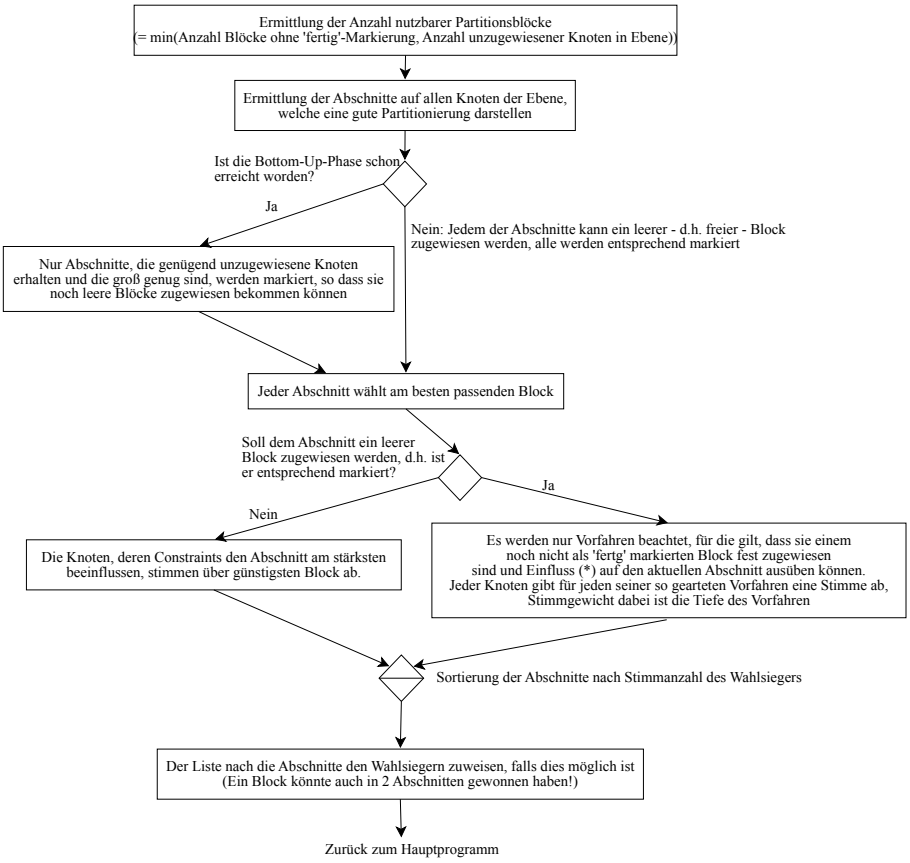

Abbildung 5.16: Flussdiagramm für die 2. Runde der Entscheidungsphase, (*): Mit Einfluss ist hier die in Abbildung 5.13 beschriebene Eigenschaft gemeint

5.4.3 Runde 3: Wahl der restlichen Prozessoren

Die grundlegende Vorgehensweise in der 3. Runde ist sehr viel einfacher als die der ersten beiden Runden. Die schon ermittelten Zonen für eine günstige Partitionierung des Entscheidungstupels müssen jetzt den restlichen Blöcken zugeordnet werden, wobei nun freie Auswahl zwischen allen Blöcken besteht, die noch nicht als `fertig` markiert wurden.

Dafür werden Informationen über die Prozessoren der einzelnen Blöcke und deren Kommunikationskapazität verwendet, so dass folgender *Greedy* - Algorithmus ausgeführt werden kann:

Die berechnungsaufwändigste Zone wird dem Block mit der größten Rechenkapazität zugewiesen. Danach wird für eine benachbarte Zone k ein noch nicht als `fertig` markierter Prozessor p gewählt.

Dafür werden, ähnlich wie in Runde 1, die Berechnungskosten und -kapazität sowie die Kommunikationskosten und -kapazität auf die Gesamtkosten aller unzugewiesener Knoten der Ebene sowie die insgesamt noch zur Verfügung stehenden Ressourcen projiziert. Dies geschieht für jeden Prozessor, so dass am Ende der Prozessor für Zone k gewählt wird, der unter den noch vorhandenen Prozessoren am besten zu den Kommunikations- und Berechnungsanforderungen der Knoten aus Zone k passt.

Hier wird also lediglich der relative Unterschied zwischen Anforderungen und vorhandenen Ressourcen betrachtet, was keine optimale, aber eine handhabbare Lösung darstellt.

5.4.4 Ende der Entscheidungsphase

Diese Phase ist recht aufwändig und komplex, vor allem weil so viele Sonderfälle betrachtet werden müssen. Sie ist aber trotzdem verhältnismäßig schnell zu berechnen, weil alle benötigten Daten schon durch die Attributierung des Baumes oder während der Top-down - Phase des Algorithmus berechnet werden können.

Das Ergebnis der Entscheidungsphase ist ein bis einschließlich zur Entscheidungsebene partitionierter Baum, bei dem die Constraints erhalten und Hardwaretopologie sowie Berechnungskosten der einzelnen Modellelemente berücksichtigt wurden. Nun müssen noch, in der Bottom-up - Phase, die weiter oben gelegenen Knoten partitioniert werden.

Der Aufbau dieser Phase erweckt vielleicht den Anschein, in Runde 3 würde nur noch ein kleiner Rest der Knoten bearbeitet, dies ist jedoch nicht der Fall. Bei gänzlich unbeschränkten Partitionierungen wird *ausschließlich* die dritte Runde (sowie die Ermittlung der besten Aufteilung, wie in Abschnitt 5.4.2 beschrieben) ausgeführt, und da es selbst bei beschränkten Partitionierungen meist nur sehr wenige Constraints gibt, wird in der letzten Runde die Mehrheit aller Knoten der Entscheidungsebene partitioniert.

5.5 Bottom-up - Phase

Ziel der letzten Phase des Algorithmus ist es, die Knoten oberhalb der Entscheidungsebene zu partitionieren. Dabei wird ebenenweise von unten nach oben vorgegangen, so dass jeweils eine bereits partitionierte Ebene unterhalb der aktuellen Ebene berücksichtigt werden kann.

Die Knoten jeder Ebene werden der Reihe nach analysiert. Gibt es bereits ein Constraint für den entsprechenden Knoten, ist die Entscheidung einfach, er wird dem geforderten Partitionsblock zugewiesen.

Andernfalls entscheiden die Blockzuweisungen seiner Kinder über den günstigsten Block. Dadurch werden große Teilbäume komplett einem Block zugewiesen, was die Kommunikationskosten verringert.

Die verschiedenen Teile der Entscheidungsebene wurden - wenn dies nicht durch Beschränkungen verhindert wurde - schon so gewählt, dass Knoten verschiedener Blöcke erst möglichst weit oben im Baum auf Vorfahren aus einem anderen Block treffen. So ist auch die Unterteilung der Entscheidungsebene gemäß der Distanzen zwischen den Knoten zu motivieren. Ohne diese Vorgehensweise der Bottom-up - Phase wäre die Nutzung der Distanzen zur Partitionierung unsinnig.

Hat ein Knoten Kinder aus verschiedenen Blöcken, kann man den Sachverhalt auf zweierlei Weise diskutieren: Man könnte den Knoten dem Block zuweisen, dem die Mehrheit der Kindknoten zugeordnet ist, um die Kommunikation gering zu halten. Andererseits könnte man den Knoten dem Block aus den Blöcken der Kinder zuweisen, der bisher am wenigsten ausgelastet ist. Dies würde unter Umständen die Kommunikationslast erhöhen, aber die Rechenlast gerechter verteilen.

Da der Algorithmus, wie in Abschnitt 3.2.1 beschrieben, vor allem die Kommunikationskosten minimieren soll, wird erstere Vorgehensweise bevorzugt. Nur bei einer uneindeutigen Auswahl wird auf das zweite Kriterium zurückgegriffen.

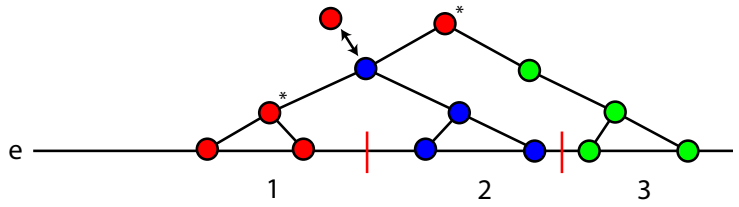

Abbildung 5.17: Knoten mit fester Zuordnung sind mit einem * gekennzeichnet. Gewinnt der blaue Block die Wahl auf der zweiten Ebene, so gibt es insgesamt drei Kanten, die zwischen Knoten verschiedener Blöcke liegen. Gewinnt dagegen der rote Block die Wahl (da der Elternknoten des Knotens auch im roten Block liegen muss), so gibt es nur 2 Kanten zwischen Blöcken, und damit weniger Kommunikation zwischen ihnen .

Ein Sonderfall entsteht dann, wenn der Knoten Vorfahren besitzt, die Beschränkungen unterliegen (siehe Abbildung 5.17). Diese Situation ist denkbar, da jedes Modellelement einzeln mit einem Constraint versehen werden kann. In diesem Fall werden für die Blockwahl des Knotens auch die Constraints seiner Vorfahren mit einbezogen. Wie stark diese die Wahl des Blocks beeinflussen, kann dabei über den Parameter $bonus_{uConstr}$ justiert werden (siehe Tabelle 5.1 auf Seite 78).

Wie bereits erwähnt, kann auch der Fall auftreten, dass ein Blatt des Baumes oberhalb der Entscheidungsebene liegt. Wenn dies geschieht, wird die Bottom-up Phase unterbrochen, und die Entscheidungsphase wird mit der Liste aller auf

dieser Ebene befindlichen Blätter erneut aufgerufen. Allerdings müssen dabei bestimmte Unterschiede zum ersten Durchlauf beachtet werden, beispielsweise wird die `fertig` - Markierung aller Blöcke zurück gesetzt. Nur dadurch ist es möglich, auch höher gelegene Ebenen noch günstig zu partitionieren.

5.6 Fazit

Obwohl die eigentliche Grundidee des Algorithmus extrem einfach und einleuchtend ist, gab es doch sehr viele Sonderfälle und Anforderungen, die eine Entwicklung erschwerten. Vor allem die Berücksichtigung von Constraints, so wichtig sie ist, trug erheblich zu diesem Sachverhalt bei.

Der selbe Algorithmus ohne dieses Merkmal würde aus höchstens der Hälfte der dafür benötigten Programmzeilen bestehen. Außerdem lässt sich im Kapitel 6 recht gut beobachten, wie die Qualität der Partitionierungsergebnisse mit steigender Anzahl von Constraints rapide nachlässt. Wenn aber - wie meistens der Fall - nur wenige Constraints festgelegt werden, so kann der Algorithmus darauf recht gut reagieren.

Davon abgesehen wird eine effiziente, wenn auch recht komplexe Vorgehensweise genutzt, um eine möglichst wenig fragmentierte Partition zu erhalten.

Insgesamt bietet der Algorithmus folgende Merkmale:

- Beachtung der Topologie des Prozessornetzwerks und der Berechnungskosten der Modellelemente, soweit dies möglich ist
- Beachtung von Constraints
- Beachtung der Baumstruktur bei der Partitionierung, wodurch eine zu starke Fragmentierung des Modellgraphen verhindert wird
- Durch Kombination aus Top-down und Bottom-up - Verfahren wird versucht, die am besten geeignete Menge an Knoten zur Partitionierung des Baumes zu analysieren
- Die Attribute der einzelnen Elemente des Modells, die während der Vorstufe hinzugefügt werden, können von anderen JAMES II - Komponenten benutzt werden, um unnötige Analysen des Modells zu vermeiden. Eine weitere Analyse des kompletten Modells sollte nach der Partitionierung nicht mehr erforderlich sein.
- Ein Modell wird nicht unnötig fragmentiert, nur weil eine genügend große Anzahl Prozessoren existiert.

Bei der recht umfangreichen Beschreibung des Algorithmus wurden die Parameter genannt, mit denen er gezielt an die praktischen Erfordernisse angepasst werden kann. Zur besseren Übersicht sind in Tabelle 5.1 noch einmal alle vorhandenen Parameter vorgestellt.

Natürlich birgt das Verfahren auch einige Nachteile, die vor allem der heuristischen Natur des Algorithmus geschuldet sind. Als Beispiel sei hier das Modell in Abbildung 5.18 angegeben.

Außerdem bietet der Algorithmus trotz seiner Komplexität noch viele Verbesserungsmöglichkeiten, zum Beispiel:

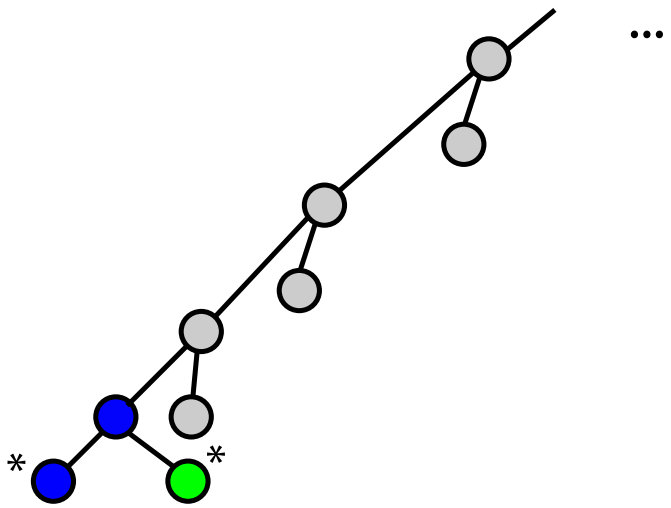

Abbildung 5.18: Beispiel für einen ungünstigen Modellbaum: Wegen den durch Constraints zugewiesenen Knoten wird die unterste Ebene als Entscheidungsebene gewählt. Bei falsch eingestellten Parametern werden alle weiteren Ebenen ebenfalls dem blauen Block zugewiesen

- Durch intelligentere Berechnung von Distanzen könnte die ebenenweise Verarbeitung des Baumes eventuell vermieden werden, so dass ein dem Ansatz von Zeigler et al. [14] ähnliches Verfahren dabei entsteht
- Unbestimmte Constraints sollten separat behandelt werden
- In der ersten Runde der Entscheidungsphase wird der projizierte Berechnungsanteil der betrachteten Prozessors als Obergrenze gewählt. Dies ist ein Aspekt, an dem beispielsweise neue Parameter eingefügt werden könnten, um dieses Kriterium den Erfordernissen anzupassen.
- Die Linearisierung der Knoten in einer Ebene ist zwar der Einfachheit zuträglich, jedoch nicht ganz korrekt. Vielmehr sollten an Stelle der Nachbarn eines Knotens jeweils alle Knoten der Ebene mit minimaler Distanz zum ihm betrachtet werden, da die Linearisierung der Ebene willkürlich vorgenommen wurde.

Abschließend sollte noch einmal klar gestellt werden, dass dieser Algorithmus natürlich für alle hierarchischen Modelle - also Modelle, die durch einen Modellbaum repräsentiert werden können - nutzbar ist. Wegen den auf die in 3.2.1 erarbeiteten Anforderungen zugeschnittenen Teilalgorithmen ist die Leistungsfähigkeit des Algorithmus bei der Partitionierung anderer Modellarten jedoch erst noch zu zeigen.

Da das ausführliche Testen aller Parameter den Rahmen dieser Arbeit übersteigt, wurden lediglich einige Stichproben zu ihrer Sensitivität untersucht. Diese werden in Abschnitt 6.1 kurz vorgestellt.

Name(Abschnitt)	**Bedeutung**
$idealNum_{PB}$ (5.3)	$idealNum_{PB}$ gibt an, wie viele Knoten idealerweise für jeden Partitionsblock in der Entscheidungsebene vorhanden sein sollten. Er beeinflusst also indirekt die Menge an Knoten in der Entscheidungsebene, indem er die Auswahl von höher gelegenen Ebenen mit weniger Knoten erschwert. Standardwert: 4 Name im Quelltext: `idealNumOfNodesPerPartitionBlock`
$maxDist_{nextNode}$ (5.4.1)	$maxDist_{nextNode}$ ist ein Faktor, der benutzt wird, um das Erweitern der Blöcke mit fixierten Knoten in der ersten Runde der Entscheidungsphase zu beeinflussen. Ist der nächste Knoten mehr als $maxDist_{nextNode} \cdot \emptyset_{distance}$, wobei $\emptyset_{distance}$ die durchschnittliche Entfernung aller Knoten auf der Ebene darstellt, entfernt, so wird die Erweiterung abgebrochen. Wird $maxDist_{nextNode}$ erhöht, werden also tendenziell mehr Knoten in Blöcke mit bereits fixierten Knoten aufgenommen. Standardwert: 1.5 Name im Quelltext: `distanceAcceptance`
$ratio_{undecided}$ (5.4.2)	Wenn - nachdem die Bottom-up - Phase bereits erreicht wurde - eine Menge noch unentschiedener Knoten einem neuen Block zugewiesen werden soll, dann darf der Anteil der noch unentschiedenen Knoten innerhalb der Zone nicht unter diesen Wert sinken. Erhöht man den Wert, dann werden weniger neue Blöcke für höher gelegene Teilbäume verwendet. Standardwert: 0.7 Name im Quelltext: `undecidedNodesinNewPB`
$minDist_{newPB}$ (5.4.2)	Genau wie $ratio_{undecided}$ spielt dieser Parameter bei der Entscheidung, höher gelegenen Teilbäumen einen neuen Block zuzuweisen, eine Rolle. Er beeinflusst die minimale *interne Distanz* der Knoten in der in Frage kommenden Zone. Die interne Distanz gibt an, wie hoch der von der Zone beeinflusste Teilbaum ist ($\frac{interneDistanz}{2}$). Jedem Teilbaum, dessen interne Distanz mindestens $minDist_{newPB} \cdot splitDistance_{min}$ beträgt (und der genug unentschiedene Knoten besitzt, siehe $ratio_{undecided}$), wird ein neuer Block zugewiesen, wobei $splitDistance_{min}$ die kleinste Distanz ist, wenn man die bezüglich Distanzen beste Entscheidungsebene zur Partitionierung hätte wählen können (siehe Abschnitt 5.4.2). Daher sollte immer $0 < minDist_{newPB} \leq 1$ gelten. Standardwert: 0.8 Name im Quelltext: `minNewPartitionBlockDistance`
$bonus_{uConstr}$ (5.5)	$bonus_{uConstr}$ ist ein Faktor, mit dem das Gewicht eines bereits zugewiesenen, übergeordneten Knotens bei der Wahl des Partitionsblocks eines Knotens auf Ebene *level* in der Bottom-Up - Phase eingeht. Wird $bonus_{uConstr}$ erhöht, dann wird mehr Rücksicht auf höher gelegene Constraints genommen. Es wird immer nur das Constraint mit der größten Tiefe betrachtet. Die Tiefe des Constraints, $depth_{constr}$, wird außerdem genutzt, um das Stimmgewicht je nach Abstand zum aktuellen Knoten zu ändern. Der Einfluss höher gelegener Knoten wird mit $bonus_{uConstr} \cdot (0.5 + \frac{depth_{constr}}{level})$ berechnet und mit dem eigentlichen Wahlergebnis für den entsprechenden Block multipliziert. Dadurch spielen Constraints höher gelegener Knoten nur dann eine Rolle, wenn - wie in Abbildung 5.17 - bereits mindestens ein Kind im selben Block liegt. Standardwert: 2.5 Name im Quelltext: `upperConstraintBonus`

Tabelle 5.1: Parameter des DEVS-Partitionierungsalgorithmus

Kapitel 6

Analyse der Testergebnisse

Die in diesem Kapitel vorgestellten Testergebnisse erheben keinen Anspruch auf Vollständigkeit, vielmehr sollen sie einen Eindruck über die Stärken und Schwächen der implementierten Algorithmen vermitteln. Das Hauptaugenmerk liegt dabei natürlich auf dem Testen des DEVS-Algorithmus, da dieser nicht nur viel komplexer ist, sondern weil er eine Neuentwicklung darstellt und deshalb besonders genau geprüft werden muss.

Die entwickelten Algorithmen zur Analyse der Ergebnisse hatten gleichzeitig die Aufgabe, den DEVS-Algorithmus bezüglich der Einhaltung von Constraints zu kontrollieren. Deshalb kann zwar nicht mit hundertprozentiger Sicherheit von der ordnungsgemäßen Funktionsweise des DEVS-Algorithmus ausgegangen werden, aber er hat in mehreren tausend Läufen immer alle Constraints erfüllt und ist immer, mit einem vollständig partitionierten Baum als Resultat, terminiert.

Alle Tests wurden lediglich auf einem Computer, einem Pentium 3 (Celeron) mit 256MB Arbeitsspeicher, ausgeführt. Die angegebenen Ausführungszeiten sind daher eher als grobe Abschätzung einer oberen Schranke anzusehen.

Zum Testen des Standardalgorithmus wurden verschiedene anerkannte Benchmark-Graphen vom „Graph Partitioning Archive“ [34] verwendet. Natürlich sind die Ergebnisse des Algorithmus - verglichen mit denen der besten Algorithmen zur Partitionierung - miserabel. Der Fokus bei der Entwicklung des Standardalgorithmus lag jedoch ganz klar auf der Ausführungsgeschwindigkeit.

Zum Schluss muss noch festgestellt werden, dass eigentlich noch viele weitere Tests vonnöten wären, um alle Aspekte der Arbeitsweise der Algorithmen aufzuzeigen. Im Folgenden werden zum Beispiel keine Spezialfälle vorgestellt, welche die Reaktion des DEVS-Algorithmus auf Prozessoren mit sehr unterschiedlichen Rechenleistungskapazitäten testen. Während der Entwicklung wurde allerdings überprüft, ob der Algorithmus gängige Situationen wie

- Es liegt ein leerer Modellgraph vor
- Es gibt nur einen Knoten im Infrastrukturgraph
- Die Bisektion eines Modellbaums für zwei Prozessoren mit sehr ungleicher Rechenkapazität

erwartungsgemäß bearbeitet. Die Reaktionsfähigkeit auf speziellere Eingabedaten wurde also während der Entwicklung überprüft, nur würde die systematische Aufarbeitung aller möglichen Fälle hier zu weit führen.

Alle Testreihen wurden mit den zur Überprüfung der Experimente nötigen Daten und detaillierteren Ergebnissen auf der beiliegenden CD gespeichert. Dabei ist zu beachten, dass durch geringfügige Änderungen an den Testkomponenten die angegebenen Initialisierungswerte für den Zufallsgenerator unter Umständen *nicht* zu den exakt gleichen Ergebnissen führen. Die Testergebnisse selbst werden allerdings die gleichen Charakteristika aufweisen, die Ergebnisse sind somit reproduzierbar.

Die verwendeten Maße zum Beschreiben von Eingabedaten und Resultaten wurden bereits in Abschnitt 3.5 definiert und können dort nachgeschlagen werden.

6.1 Analyse der Parameter des DEVS-Algorithmus

Um die Auswirkungen der einzelnen Parameter objektiv zu begutachten, müsste eine große Menge verschiedener Tests durchgeführt werden, schließlich geht es dabei um fünf verschiedene Eingabeparameter(siehe Tabelle 5.1 auf Seite 78), die drei Ausgabegrößen (Schnittgröße, Ungleichgewicht und Ausführungszeit) beeinflussen.

Da dies für den Rahmen dieser Arbeit zu aufwendig erschien, aber trotzdem exemplarisch gezeigt werden sollte, wie der Einfluss der Parameter gemessen werden kann, wurden lediglich je 10 zu partitionierende Modelle mit verschiedenen Parametereinstellungen getestet.

Folgende fünf Parameterkonfigurationen wurden zum exemplarischen Testen ausgewählt:

- *Normal*: In dieser Konfiguration wurden die Standardwerte der fünf Parameter, wie sie in Tabelle 5.1 beschrieben sind, beibehalten
- *Hastig partitionieren*: Hier wurde der Parameter $idealNum_{PB}$ von 4 auf 1 gesetzt. Das bedeutet, dass bei dieser Konfiguration die Auswahl der Entscheidungsebene eventuell früher getroffen werden kann, da diese nur genauso viele Knoten umfassen muss wie die Anzahl der vorhandenen Partitionsblöcke (anstatt vier mal so viele).
- *Bessere erste Runde*: Der Parameter $maxDist_{nextNode}$ wurde bei dieser Konfiguration von 1.5 auf 10 erhöht. Damit wird praktisch sichergestellt, dass die Partitionsblöcke, die fest zugewiesene Knoten auf der Entscheidungsebene enthalten (und damit in der ersten Runde der Entscheidungsebene weiter aufgefüllt werden), wirklich so viele Knoten erhalten, wie es ihnen nach Rechenleistung zusteht - die Entfernung eines neuen Knoten zu den restlichen Knoten darf dabei statt 1.5 jetzt 10 mal höher sein als die Durchschnittsdistanz, so dass die Vermeidung von weiteren Schnittkanten hierdurch quasi unterbunden wird. Daher werden die Blöcke in der ersten Runde so mit Knoten aufgefüllt, dass dies besser ihrer Rechenkapazität entspricht.
- *Neue Blöcke*: In dieser Konfiguration wird die Zuweisung von Teilbäumen oberhalb der Entscheidungsebene zu neuen Blöcken erleichtert (siehe Abschnitt 5.4.2). Dafür müssen gleich zwei Parameter der Algorithmus modi-

fiziert werden, die das Kriterium, wann eine Zone einem neuen Block zugewiesen wird, beeinflussen: $ratio_{undecided}$ wurde von 0.7 auf 0.2 verringert, dass heißt, die in Frage kommende Zone muss nun nur noch 20% unentschiedene Knoten enthalten (statt 70%). Außerdem wurde $minDist_{newPB}$ von 0.8 auf 0.3 gesenkt, womit die größte Distanz zwischen zwei Knoten der Zone nur noch 30% von $splitDistance_{min}$ betragen muss.

- *Höhere Constraints*: Bei dieser Konfiguration wurde der Wert von $bonus_{uConstr}$ von 2.5 auf 10 gesetzt, womit höhere Constraints in der Bottom-Up Phase vier mal mehr Gewicht besitzen, wenn ein Knoten in dieser Phase einem Partitionsblock zugeordnet werden soll.

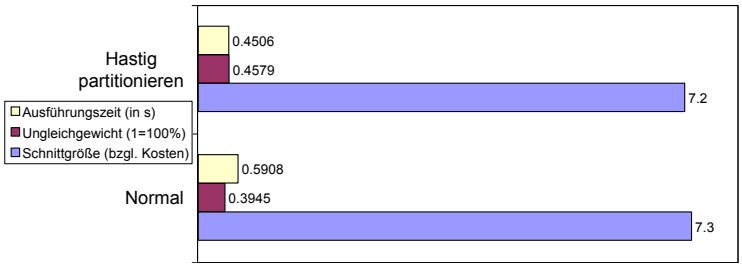

Abbildung 6.1: Vergleich zwischen normaler Konfiguration und einem „hastigeren“ Ansatz

Alle in den hier vorgestellten Testläufen festgelegten (konkreten) Constraints wurden zufällig gewählt, das heißt ein zufällig aus dem Modellgraph ausgewählter Knoten wurde einem zufällig ausgewählten Block fest zugewiesen (die Wahrscheinlichkeiten waren jeweils gleichverteilt).

Zum Testen der Konfigurationen wurden zwei Szenarien genutzt: Im ersten Szenario wurden keine Constraints verwendet, so dass die Konfiguration „Hastig partitionieren“ getestet werden konnte, ohne dass ungünstig liegende Constraints das Ergebnis verfälschen konnten (zur Erinnerung: die Entscheidungsebene muss immer so gewählt werden, dass unterhalb keine Konflikte bezüglich Constraints existieren, siehe Abschnitt 5.3).

Im zweiten Szenario wurden vier Constraints gewählt. In beiden Szenarien war die Größe der Bäume konstant 200 Knoten, die auf acht Prozessoren verteilt werden sollten, und der Verzweigungsfaktor betrug vier.

Die Ergebnisse der Tests lassen sich in den Abbildungen 6.1 und 6.2 begutachten: Dabei ist in Abbildung 6.1 deutlich zu erkennen, dass die Auswahl einer weiter oben gelegenen Entscheidungsebene, wie in der „hastigen“ Konfiguration eingestellt, die Schnittgröße leicht verringert (da weiter oben im Baum partitioniert wird), und auch die Ausführungszeit. Dies wird durch ein größeres Ungleichgewicht zwischen den Prozessoren erkauft, was damit zu erklären ist, dass höhere Ebenen (mit weniger Knoten) schlechter auf alle vorhandenen Partitionsblöcke aufteilbar sind.

In Abbildung 6.2 wird zunächst deutlich, dass die Konfigurationen „Höhere Constraints“ und „Neue Blöcke“ keinerlei Auswirkungen auf die Ergebnisse der Tests hatten - die Resultate (abgesehen von der Ausführungszeit) entsprechen genau denen der normalen Konfiguration. Das liegt daran, dass die

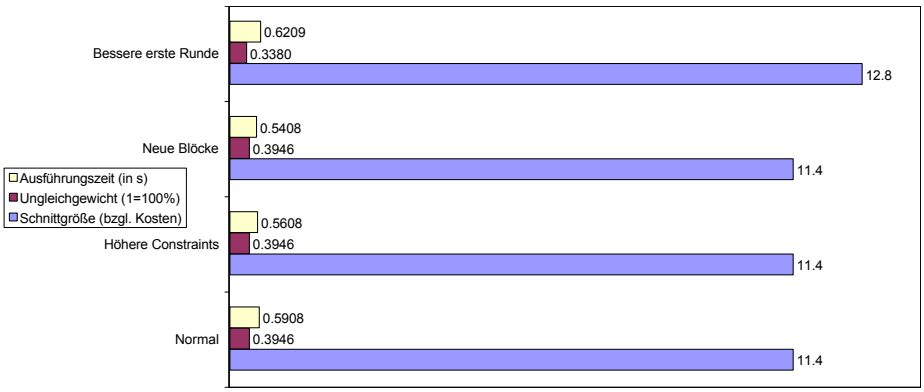

Abbildung 6.2: Vergleich zwischen normaler Konfiguration und drei weiteren Konfigurationen

Berücksichtigung höherer Constraints bzw. das Zuweisen neuer Blöcke außerhalb der Entscheidungsebene lediglich in wenigen Sonderfällen eine Rolle spielt.

Die drei Parameter $bonus_{uConstr}$, $ratio_{undecided}$ und $minDist_{newPB}$, die in den beiden Konfigurationen verändert wurden, scheinen die Ergebnisse des Algorithmus also am wenigsten zu beeinflussen.

Bei der Konfiguration „Bessere erste Runde“ ist eine Veränderung der Werte zu erkennen. Dadurch, dass die Entfernung der Knoten in der ersten Runde der Entscheidungsphase ignoriert wird, entsteht ein um 10% größerer Schnitt, wobei das Ungleichgewicht der Prozessoren um knapp 6% verringert wird. Während die Erhöhung des Schnittes und die Verringerung des Ungleichgewichts an sich nicht weiter verwunderlich sind, zeigt sich hier jedoch eine Einstellungsmöglichkeit zur Gewichtung von Ungleichgewicht bzw. Kommunikationskosten.

Dies ist ein interessanter Hinweis für die praktische Anwendung des Algorithmus, selbst wenn diese Justierung nur in einigen Fällen Wirkung zeigen kann: Voraussetzung dafür ist zum Einen das Vorhandensein von Constraints, und zum Anderen muss der Baum in den Ebenen mit Constraints genügend weitere Knoten besitzen, so dass eine Verringerung des Ungleichgewichts überhaupt möglich ist.

6.2 Testegebnisse des DEVS-Algorithmus

Die in Abschnitt 3.5 beschriebenen Testszenarien konnten mit den in 4.3 beschriebenen Komponenten automatisiert ausgeführt werden. Für die drei wichtigsten Eigenschaften der Eingabe - Anzahl der Constraints, Größe des Modellgraphen und Verzweigungsfaktor - wurden jeweils drei Testläufe durchgeführt.

Diese Testläufe, im Folgenden auch schlicht als (Versuchs-, Test-)Reihen bezeichnet, wurden mit veränderten Eigenschaften der Eingabedaten ausgeführt, um weitere Zusammenhänge zu erkennen.

In den drei Testläufen wurden die beiden anderen Variablen oder die Anzahl der Prozessoren variiert, so dass man einen Eindruck über die unterschiedlichen Abhängigkeiten der Parameter gewinnen kann. Pro Lauf wurde der DEVS-Algorithmus je 200 mal mit wechselnden Eingaben aufgerufen, insgesamt wurde

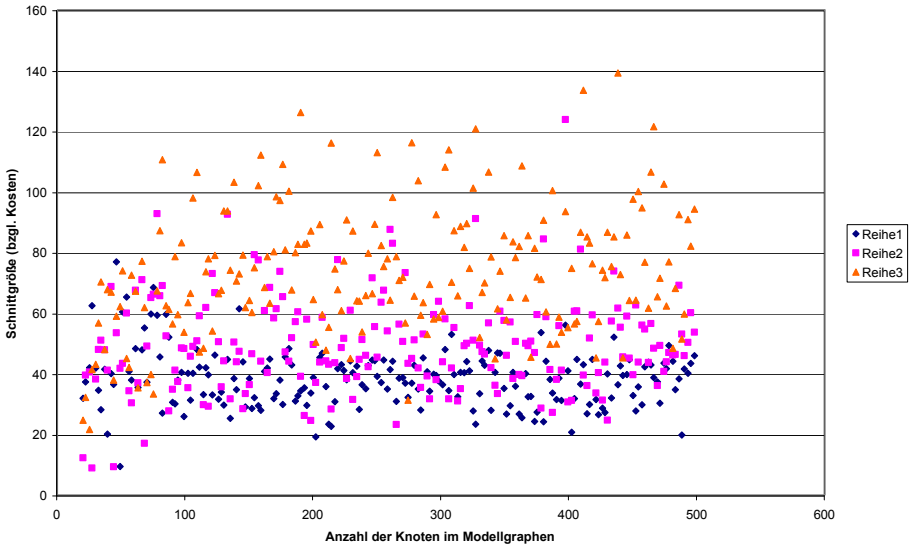

Abbildung 6.3: Schnittgröße in Abhängigkeit von der Größe des Modellgraphen. Die Schnittkosten werden nicht von der Größe des Modellgraphen beeinflusst, jedoch stark von Anzahl und Lage der Constraints (daher die hohe Streuung bei der dritten Reihe).

er damit mehr als 1800 mal getestet.

An dieser Stelle sollen exemplarisch die Testergebnisse zum Einfluss der Größe des Modellgraphen auf die Ergebnisse analysiert werden, alle weiteren Testergebnisse mit kurzen Erläuterungen sind im Anhang C nachzulesen.

Wie auch bei der Untersuchung der anderen beiden Charakteristika der Eingabedaten wurden drei verschiedene Versuchsreihen durchgeführt: Allen gemeinsam ist die Aufteilung des Modellgraphen auf acht Partitionsblöcke und der Verzweigungsfaktor des Graphen, der vier beträgt. In der ersten Versuchsreihe wurden keinerlei Constraints festgelegt, das heißt die Entscheidungsphase beschränkte sich auf die dritte Runde. In der zweiten Reihe wurden jeweils vier Constraints zufällig gewählt, so dass es relativ wenige Constraints im Graphen gab. In der dritten und letzten Reihe wurden schließlich acht Constraints festgelegt, so dass eine Konfliktsituation und damit die Nutzung einer der Heuristiken für die diversen Sonderfälle wahrscheinlicher wurde.

Die Abbildung 6.3 zeigt eindrucksvoll, dass die Größe des Graphen die Schnittkosten ab einem gewissen Wert nicht mehr beeinflusst - ganz im Gegensatz zu den Constraints. Da das Hauptaugenmerk bei der Entwicklung des Algorithmus auf der Vermeidung von Kommunikation zwischen den Prozessoren lag, ist dieses Ergebnis besonders befriedigend. Es ist auf die hierarchische Herangehensweise und die Nutzung der Baumeigenschaft zurückzuführen. Es ist außerdem zu beachten, dass die Schnittkosten bei Partitionierungen ohne Constraints nicht sehr stark variieren. In der Tat konnte bei den Versuchen festgestellt werden, dass der Schnitt meistens nur eine nahezu minimale Anzahl der Kanten umfasste, so dass die Streuung in der Testreihe eins vor allem auf die zufällig gewählten Kantenbeschriftungen im Modellgraphen zurückzuführen ist.

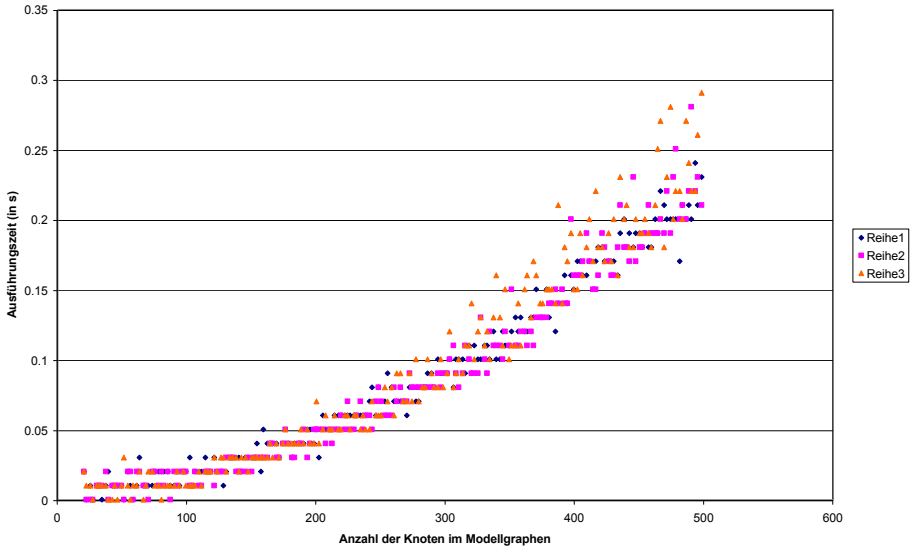

Abbildung 6.4: Ausführungszeit in Abhängigkeit von der Größe des Modellgraphen. Hier ist zu erkennen, dass die Ausführungszeit bei Vergrößerung des Eingabegraphen stark ansteigt. Dies könnte jedoch nicht nur durch die Vorgehensweise des Algorithmus selbst, sondern auch durch mögliche Schwächen in der Implementierung verursacht werden.

In Reihe eins wurden 167 Resultate für Modellgraphen mit mehr als 100 Knoten ermittelt, davon benötigten 111 (ca. 66%) die minimale Anzahl von sieben Kanten zur Partitionierung auf acht Prozessoren. Bei kleineren Modellgraphen ist das Ergebnis stärker von der konkreten Struktur abhängig. Das alles deutet darauf hin, dass der Algorithmus in vielen (unbeschränkten) Fällen eine Partition findet, die einen sehr geringen Kommunikationsaufwand erfordert.

In Abbildung 6.4 wird jedoch deutlich, dass die Zeitkomplexität des Algorithmus noch zu wünschen übrig lässt, da ein starker Anstieg beobachtet werden kann. Dieses Verhalten könnte möglicherweise durch eine effizientere Implementierung verbessert werden.

In Abbildung 6.5 ist zu sehen, dass zwar vor allem die Partitionen von kleineren Graphen unausgewogen sind, aber dass die Vermeidung von Ungleichgewicht eine generelle Schwäche des Algorithmus ist. Selbst ohne Constraints beträgt die durchschnittliche Differenz zwischen der relativen Last und der relativen Rechenleistung ca. 40%, was natürlich enorm ist. Allerdings muss dazu gesagt werden, dass die Verwendung von zufälligen Constraints hier einen falschen Eindruck erwecken kann: Gerade *weil* der Algorithmus vor allem die Kommunikationskosten minimiert, können Constraints in der Praxis dazu eingesetzt werden, bestimmte Teile des Modells explizit bestimmten Prozessoren zuzuweisen - das heißt, es besteht die Möglichkeit, dass die bewusste anstatt zufällige Verwendung von Constraints das Ungleichgewicht eines Ergebnisses drastisch verringern kann.

Des Weiteren konnte durch das Testen festgestellt werden, dass die Struktur des Modellgraphen, bestimmt durch den Verzweigungsfaktor, sehr wenig

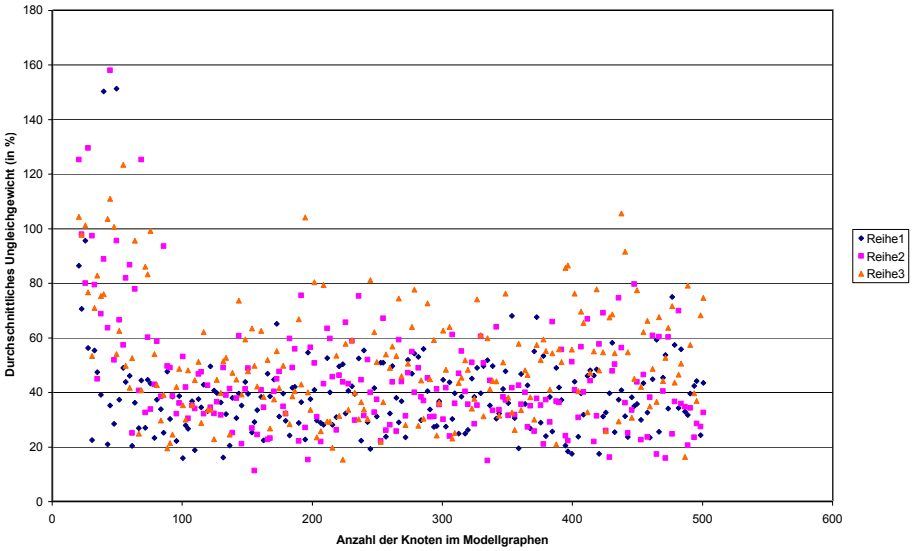

Abbildung 6.5: Ungleichgewicht in Abhängigkeit von der Größe des Modellgraphen: Abgesehen von Modellgraphen, die zu klein für eine gerechte Aufteilung sind, scheint die Größe des Graphen keinen Einfluss auf das Ungleichgewicht zwischen den Prozessoren zu nehmen.

Einfluss auf die Ergebnisse nimmt. Anders die Anzahl der Constraints, die vor allem die Streuung der Ergebnisse erhöht. Diese Ergebnisse waren jedoch wenig erstaunlich, so dass die genauen Details hier ausgespart werden. Die entsprechenden Diagramme sind in Anhang C zu finden.

Zum Abschluss wurde noch die Partitionierung von besonders großen Modellbäumen getestet, da dies relevant für den praktischen Einsatz des Algorithmus ist. Zu den Ergebnissen, die in Abbildung 6.6 zu sehen sind, muss allerdings gesagt werden, dass die verwendete Hardware dieser Aufgabe nicht gewachsen schien, so dass die angegebenen Ausführungszeiten eventuell durch *Swapping*-Operationen des Betriebssystems etc. nach oben verfälscht wurden.

6.3 Testegebnisse des Standardalgorithmus

Im Vergleich zum DEVS-Partitionierungsalgorithmus wurde der Standardalgorithmus nur sehr wenig getestet. Er hat zuallererst die Aufgabe, als eine Art Notfall-Algorithmus alle Modelle, für die keine speziellen Algorithmen existieren, zu partitionieren. Daher wurde der Algorithmus vor allem hinsichtlich Stabilität bei großen und strukturell unbekannten Modellgraphen getestet. Die Ergebnisse des Tests sind in Tabelle 6.3 zusammengefasst.

Die Angabe des Ungleichgewichts wurde dabei ausgelassen, weil es in allen Fällen extrem gering war. Das liegt an der Vorgehensweise des Algorithmus, der ja im Grunde nur eine geordnete Liste von Knoten zerteilt. Es kann also der Einfachheit halber davon ausgegangen werden, dass der Algorithmus die Last optimal den vorhandenen Prozessoren zuweist. Dabei ist allerdings zu beachten,

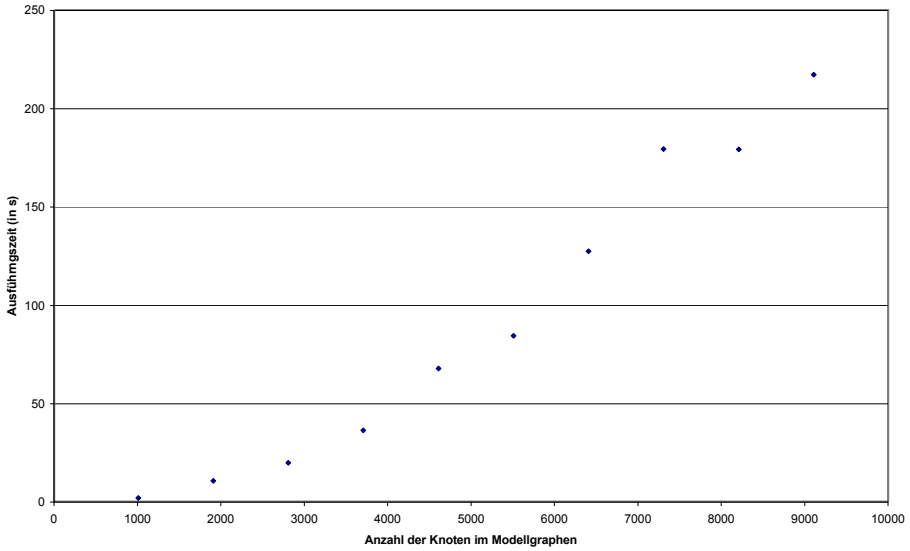

Abbildung 6.6: Ausführungszeiten des DEVS-Algorithmus bei besonders großen Modellgraphen: Getestet wurden Bäume mit 10 Constraints und einem Verzweigungsfaktor von 10. Die Graphen wurden auf 16 Partitionsblöcke aufgeteilt.

Name des Modellgraphen (siehe [34])	**Anzahl der Knoten**	**Anzahl der Kanten**	**Ausführungszeit** (in s)	**Schnittgröße** (Anzahl der Kanten)
`data`	2851	15093	0.8211808	312
`memplus`	17758	54196	49.9017552	16746
`wing`	62032	121544	3.8255008	4776

Tabelle 6.1: Testergebnisse des Standardalgorithmus: Jeder Graph wurde bisektioniert

dass die Schnittkosten dementsprechend hoch sind.

Die lange Ausführungszeit zur Partitionierung des Graphen `memplus` wird durch die interne Struktur des Modellgraphen hervorgerufen. Er enthält einzelne Knoten mit mehreren hundert Nachbarn, so dass deren Sortierung in diesem Fall sehr aufwendig war.

Wie dem auch sei, die Tabelle belegt, dass der Algorithmus auch für sehr große Graphen schnelle Ergebnisse liefert, womit er seine Aufgabe im Framework gut erfüllt.

Zum Abschluss der Testanalyse soll hier noch der Unterschied zwischen dem Standardalgorithmus und dem DEVS-Partitionierungsalgorithmus in einem direkten Vergleich veranschaulicht werden. Dazu wurde ein Baum verwendet, da der DEVS-Algorithmus keine anderen Graphen als Eingabe akzeptiert. Die Anzahl der Knoten betrug 200, der Verzweigungsfaktor war 4 und es wurden keine Constraints festgelegt, da der Standardalgorithmus keine Constraints berücksichtigt. Das Ergebnis dieses „Tests“ ist in Abbildung 6.7 zu begutachten.

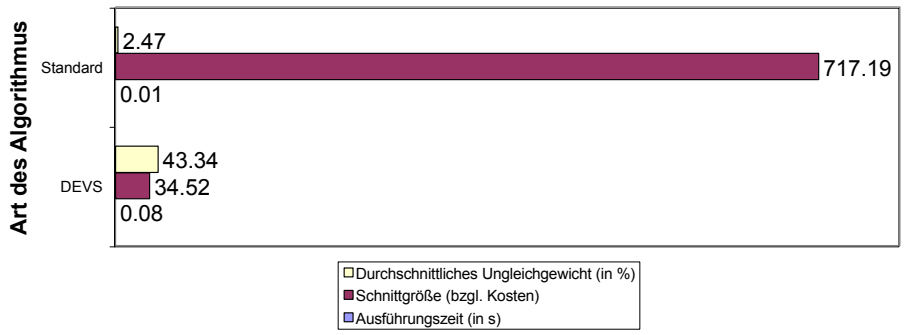

Abbildung 6.7: Vergleich zwischen Standard- und DEVS-Algorithmus: Der Standardalgorithmus minimiert das Ungleichgewicht, der DEVS-Algorithmus versucht, die Kommunikationskosten zu minimieren.

Kapitel 7

Zusammenfassung

Wissenschaft nennen wird jenen kleinen Teil der Unwissenheit, den wir geordnet und klassifiziert haben.
Ambrose Bierce

7.1 Auswertung

Die Zielstellung dieser Arbeit war, einen Mechanismus zur Modellpartitionierung zu entwickeln und diesen für JAMES II zu implementieren. Nachdem eine grundlegende Einführung in die Thematik (Kapitel 1) nebst kurzer Vorstellung vorhandener Methoden (Kapitel 2) erfolgte, wurden die Anforderungen an den Mechanismus sowohl aus softwaretechnischer als auch aus algorithmischer Perspektive erarbeitet (Abschnitt 3.2).

Basierend auf den Überlegungen in Kapitel 3 wurde ein Framework zur Modellpartitionierung in JAMES II implementiert, dass in Kapitel 4 näher beschrieben wurde. Es kann, zusammen mit einer automatisch erzeugten Dokumentation des Quelltextes, auf der beiliegenden CD gefunden werden.

Die wichtigste der in Kapitel 3 begründeten Hauptthesen besagt, dass eine optimale Modellpartitionierung immer eine gewisse Spezialisierung auf eine Teilklasse von Graphen bzw. Modellierungsformalismen erfordert. Daher wurde das Framework so offen und erweiterbar wie möglich gestaltet, und es wurden drei verschiedene Algorithmen implementiert.

Während sich der KL-Algorithmus (Abschnitt 2.2) später als zu langsam herausstellte und nur als Standardalgorithmus für sehr kleine Graphen genutzt wird, und sein Ersatz zur Partitionierung von beliebigen Graphen (Abschnitt 4.1) auf Grund der hohen Geschwindigkeitsanforderungen so trivial wie möglich gehalten werden musste, lag das Hauptaugenmerk im zweiten Teil dieser Arbeit auf der Entwicklung eines geeigneten Partitionierungsalgorithmus für die in JAMES II vornehmlich verwendeten DEVS-Modelle (Kapitel 5).

Da der Algorithmus möglichst viele der in Kapitel 3 formulierten Anforderungen erfüllen sollte, insbesondere die Minimierung der Kommunikationskosten und die Berücksichtigung von Constraints, wurde der Algorithmus komplexer als zunächst erwartet. Ausgehend von der in [14] skizzierten Grundidee der hierarchischen Partitionierung von DEVS-Modellen, stützt er sich bei der Partitionierung der Modelle auf eine Vielzahl von Heuristiken, die zwar alle begründet,

aber nicht bewiesen wurden. Aus diesen Gründen wurde er ausführlich analysiert und getestet (siehe Kapitel 6).

Außerdem sind viele der im Algorithmus genutzten Verfahren zwecks Minimierung der Kommunikation entwickelt worden, der Algorithmus wird daher nicht immer *insgesamt* gute Ergebnisse liefern. Das eigentliche Ziel aber, einen auf DEVS-Modelle spezialisierten Algorithmus mit bestimmten Merkmalen zu entwickeln, wurde erreicht. Der Algorithmus selbst ist also eher als eine Art Machbarkeitsstudie anzusehen, und soll die These untermauern, dass die Entwicklung dieser Art von spezialisierten Partitionierungsalgorithmen ein sinnvoller Weg ist, um die Qualität der Ergebnisse bezüglich gewisser Kriterien zu erhöhen.

Alles in allem erfüllen aber sowohl der DEVS-Algorithmus als auch das Framework selbst recht gut die an sie gestellten Bedingungen. Der Algorithmus arbeitet stabil und ausreichend effizient, er beachtet Constraints und versucht, die Kommunikationskosten zu minimieren. Allerdings werden die so partitionierten DEVS-Modelle recht ungleichmäßig auf die vorhandenen Prozessoren verteilt, was sicherlich einen Nachteil darstellt. Wie stark sich dieser Nachteil hinsichtlich der Vorteile des Algorithmus auswirkt, und wie sich die Qualität der Ergebnisse im Vergleich zum in Abschnitt 2.5 beschriebenem Ansatz verhält, muss noch untersucht werden. In [15] bestätigen Zeigler et al. zwar die gute Leistungsfähigkeit vom Algorithmus in Bezug auf Ungleichgewicht, es wurden jedoch keinerlei Aussagen über die Schnittgröße der erzielten Resultate getroffen (siehe Abschnitt 2.5).

Die Vor- und Nachteile des Frameworks selbst lassen sich nur schwer abschätzen, da viele Nachteile erst bei der Implementierung von weiteren Ansätzen erkennbar sein könnten. Grundsätzlich wurde aber versucht, eine abstrakte Herangehensweise zu nutzen, um eine breite Vielfalt von Partitionierungsalgorithmen zuzulassen. Dies schließt ausdrücklich die Unterstützung von zukünftigen Load-Balancing-Algorithmen und deren nahtlose Integration ins Framework mit ein. Das hier beschriebene System ist funktionstüchtig und wurde bereits testweise in JAMES II integriert.

Die Einordnung dieser Ergebnisse in die vorhandenen Ergebnisse auf dem Gebiet der Partitionierung fällt sehr schwer, da es sich hierbei um ein heterogenes Forschungsgebiet handelt. Dies liegt vor allem daran, dass die Mehrzahl der entwickelten Algorithmen nicht auf Bäumen, sondern auf Graphen arbeitet. Hinzu kommt die Konzentration auf die Partitionierung von DEVS-Modellen in dieser Arbeit. Der einzige ähnliche Ansatz ist daher die Methode der DEVS-Partitionierung aus Abschnitt 2.5, wobei sich noch zeigen muss, ob die hier vorgeschlagene Vorgehensweise eine Verbesserung darstellt. Die Partitionierung von DEVS-Modellen unter Berücksichtigung von Constraints ist allerdings ein neues Merkmal für DEVS-Partitionierungsalgorithmen - dessen Relevanz jedoch nur schwer eingeschätzt werden kann.

7.2 Ausblick

Neben den zahlreichen möglichen Verbesserungen des DEVS-Algorithmus (siehe Abschnitt 5.6) und einer verbesserten Implementierung, sowie der genauen Untersuchung seiner Parameter, sollten vor allem die Vorstufen des Algorithmus - also die Analyse der vorhandenen Infrastruktur und des zu simulierenden

Modells - intensiv bearbeitet werden.

Schließlich ist die Aussagekraft der Eingabedaten eine notwendige Voraussetzung für eine gute Partitionierung der Modelle, und die Erfassung dieser Daten ist mit Sicherheit keine triviale Aufgabe.

Davon abgesehen gibt es natürlich eine Vielzahl anderer Modellierungsformalismen, wie beispielsweise StateCharts oder zelluläre Automaten, für die entsprechende Modellanalyse- und Partitionierungskomponenten entwickelt werden könnten. Denkt man hier etwas weiter in die Zukunft, könnte man die Entwickler dieser Komponenten durch die Entwicklung einer Bibliothek mit Standardkomponenten zur Lösung häufig auftretender Probleme entlasten.

Eine andere interessante Entwicklungsmöglichkeit bietet die Implementierung eines visuellen Werkzeugs, dass die Eingabe der Constraints durch den Modellierer vereinfachen würde. Gleichzeitig könnte der Modellierer damit einige Parameter des verwendeten Algorithmus verändern, um dann interaktiv die Qualität der Partitionierung zu begutachten und gegebenenfalls einfach von Neuem zu partitionieren, mit veränderten Parametern oder einem anderen Algorithmus. Treibt man diese Idee auf die Spitze, wäre die Integration so eines Werkzeugs in einen Dialog zum Starten der Simulation innerhalb der James II - Oberfläche denkbar. Ähnlich wie bei einem Dialog zum Drucken oder zum Exportieren von Dateien, würde der Nutzer zuerst ein paar allgemeine Einstellungen in Form von Simulationsparametern vornehmen, dann die Ressourcen auswählen, die er nutzen möchte, und vor dem endgültigen Start der Simulation einen Partitionsvorschlag von einem auf diesen Modelltyp spezialisierten Algorithmus erhalten. Dieser kann nun mit intuitiven graphischen Mitteln, zum Beispiel der Zuordnung von Modellelementen zu einzelnen Partitionsblöcken per Drag & Drop, an die Wünsche des Nutzers angepasst werden. Erst danach wird die Simulation gestartet. Wegen der guten Unterstützung von interaktiven SVG-Grafiken in Java und den bereits für diese Arbeit implementierten Komponenten zur Darstellung von Graphen als SVG-Dokumente (mit Hilfe der externen *Graphviz*-Software, [36]), ist dies durchaus umsetzbar.

Zum Schluss muss noch einmal betont werden, dass die initiale Modellpartitionierung nur einen kleinen Beitrag zur effizienten verteilten Ausführung von Simulationen leisten kann. Viel wichtiger und weitaus komplexer ist dabei das schon in Abschnitt 1.2 beschriebene Problem des *Load Balancing*, also das Repartitionieren des Modells zur Laufzeit. Weil sich Kommunikations- und Rechenanforderungen einzelner Modellelemente im Laufe einer Simulation stark verändern können, müssen zu gewissen Zeiten gewisse Teile des Modells auf andere Prozessoren transferiert werden. Dabei ist zusätzlich zum Problem der Modellrepartitionierung auch die Wahl einer Zeitspanne zwischen zwei Load-Balancing Schritten ein schwerwiegender Aspekt, sowie die Entscheidung, ob sich der Transfer von bestimmten Modellelementen überhaupt lohnt. Hinzu kommt, dass der Algorithmus auf bereits vorhandenen und über diverse Prozessoren verteilten Strukturen arbeiten muss.

Ohne ein gutes *Load-Balancing* nützt jedoch die beste initiale Partitionierung wenig, denn sobald das Modell einen dynamischen Charakter entwickelt, kann die Partitionierung schon nach wenigen Zeitschritten in der Simulationszeit sehr ungünstig gewählt sein. Gerade bei der Simulation von Multiagentensystemen, eine der Aufgaben von JAMES II, spielt die Dynamik der Modelle zwecks Modellierung von Mobilität und ähnlichen Phänomenen eine zentrale Rolle.

Deshalb sollten sich die weiteren Anstrengungen zur effizienten verteilten

Simulation in JAMES II besonders auf dieses interessante und herausfordernde Thema konzentrieren.

Anhang A

UML-Diagramme der Komponenten

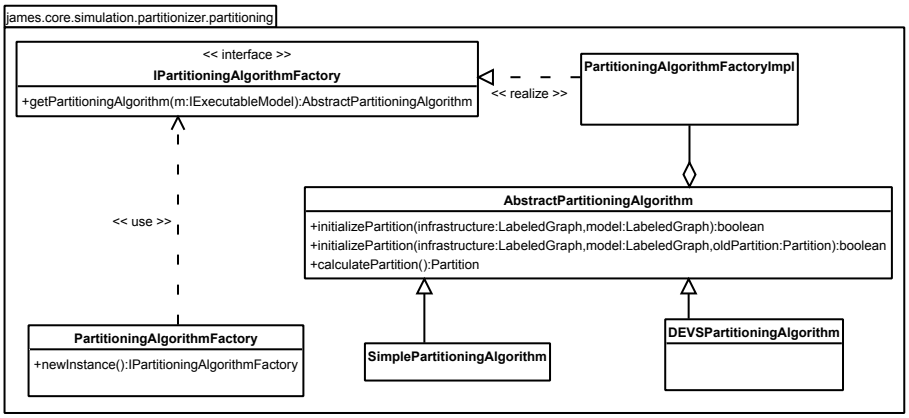

Abbildung A.1: Klassendiagramm Partitionierungsalgorithmen

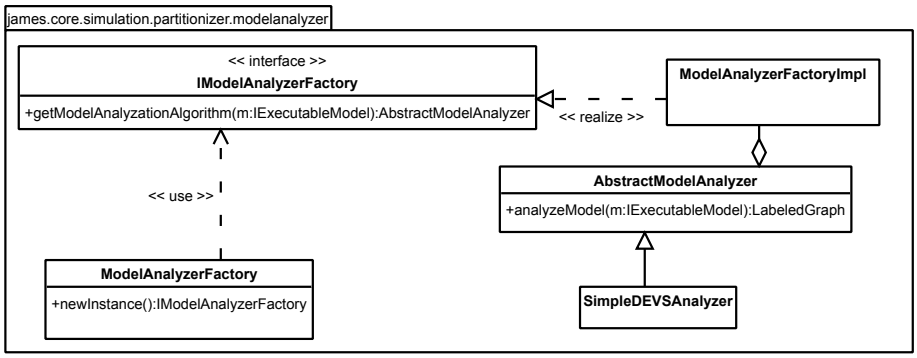

Abbildung A.2: Klassendiagramm Modellanalysealgorithmen

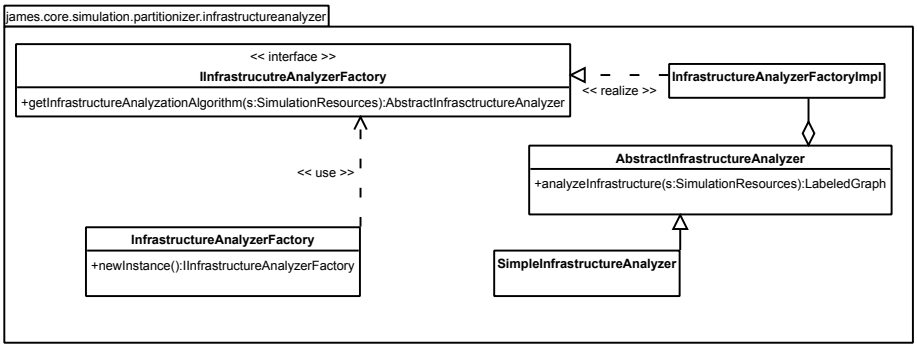

Abbildung A.3: Klassendiagramm Infrastrukturanalysealgorithmen

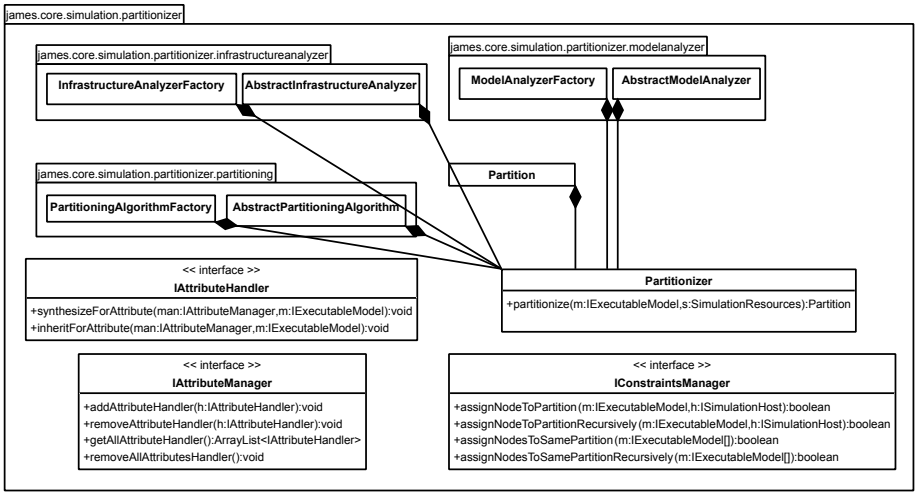

Abbildung A.4: Vereinfachtes Klassendiagramm für Partitionizer - Paket

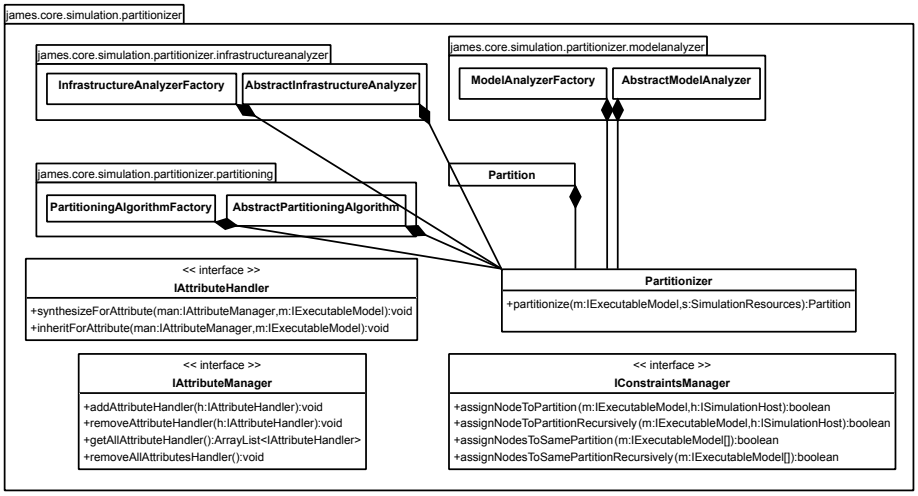

Abbildung A.5: Klassendiagramm für DEVS-Partitionierungsalgorithmus

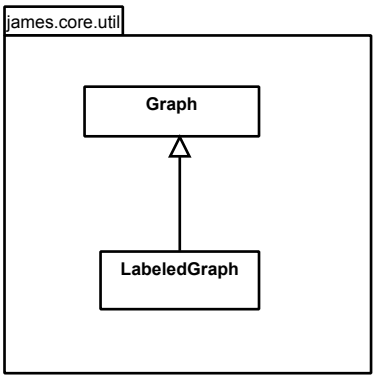

Abbildung A.6: Klassendiagramm für Graphen-Klassen

Anhang B

Flussdiagramme des DEVS - Algorithmus

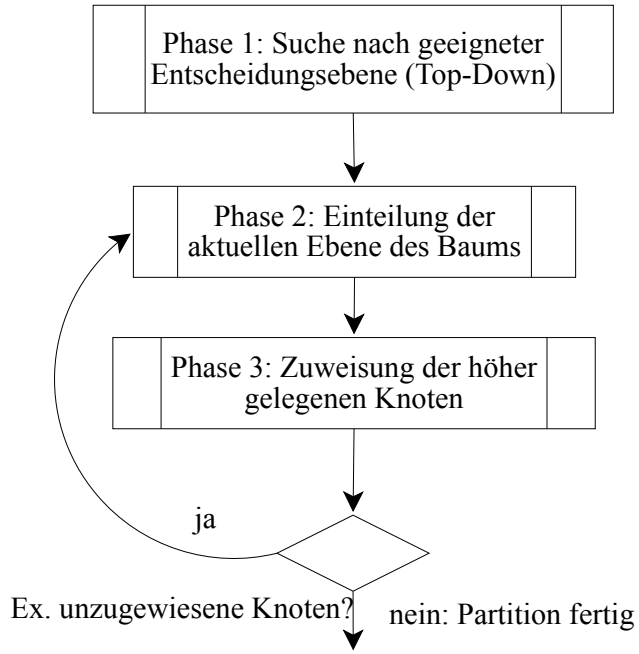

Abbildung B.1: Flussdiagramm für die Abfolge der Phasen

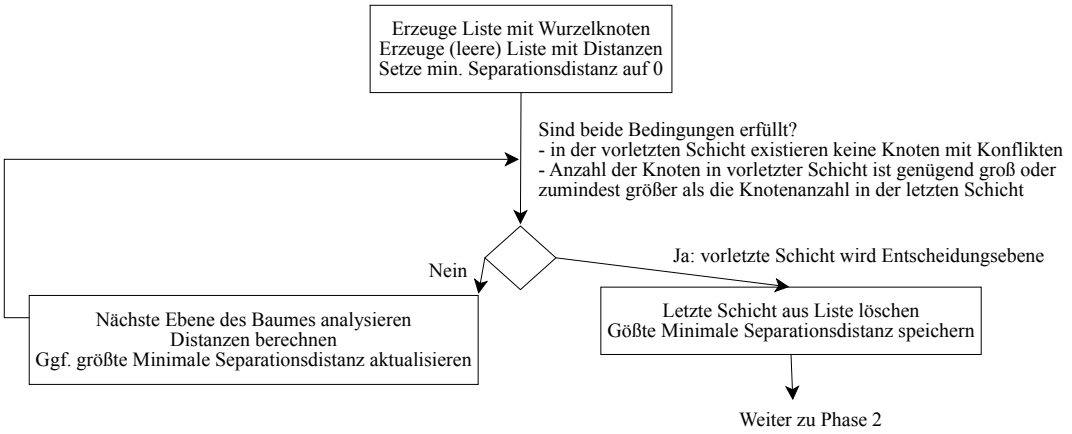

Abbildung B.2: Flussdiagramm für die Top-down - Phase

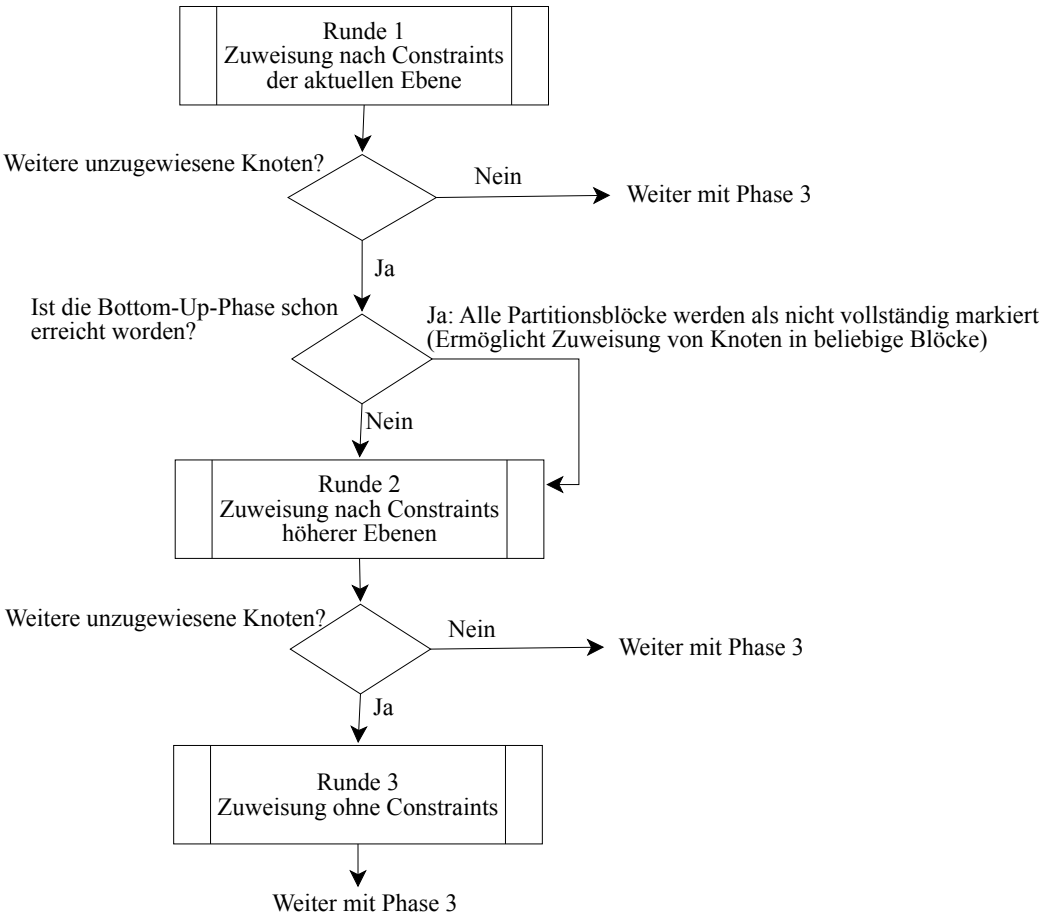

Abbildung B.3: Flussdiagramm für die Entscheidungsphase

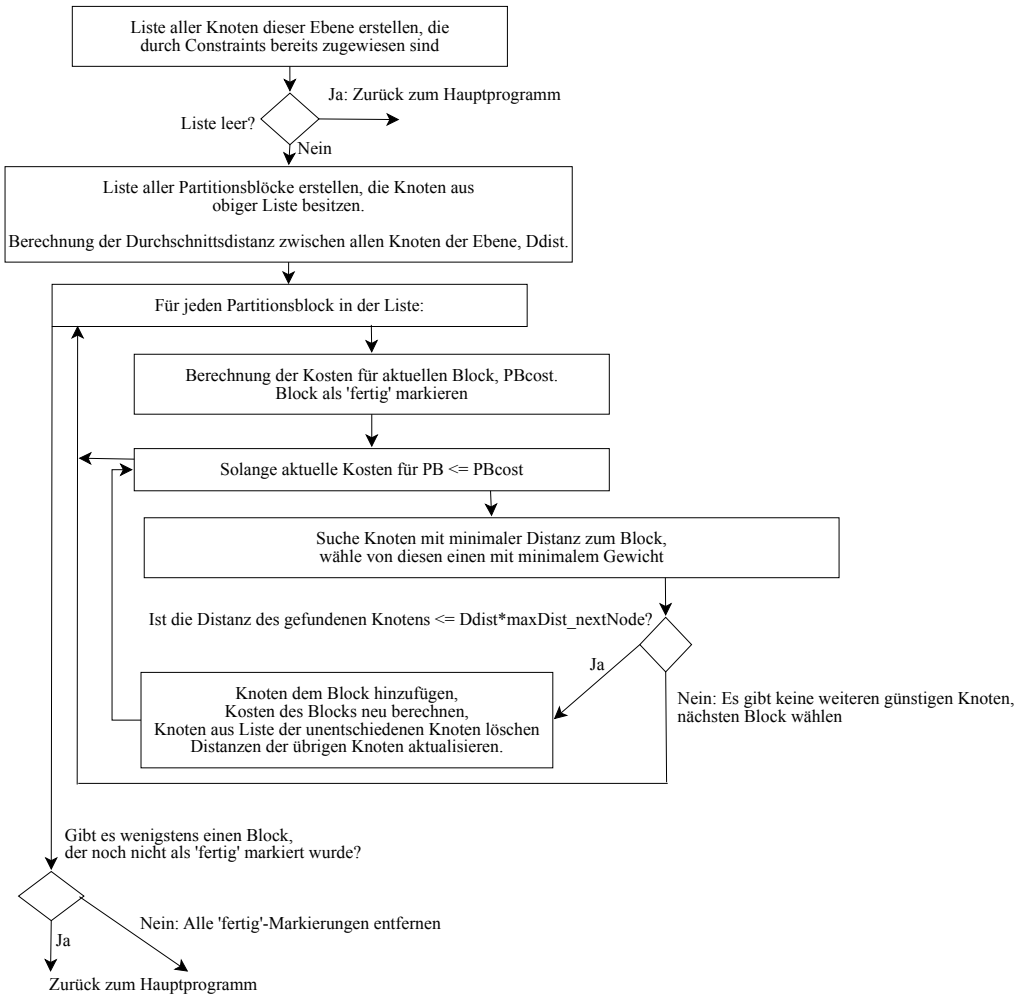

Abbildung B.4: Flussdiagramm für die 1. Runde der Entscheidungsphase

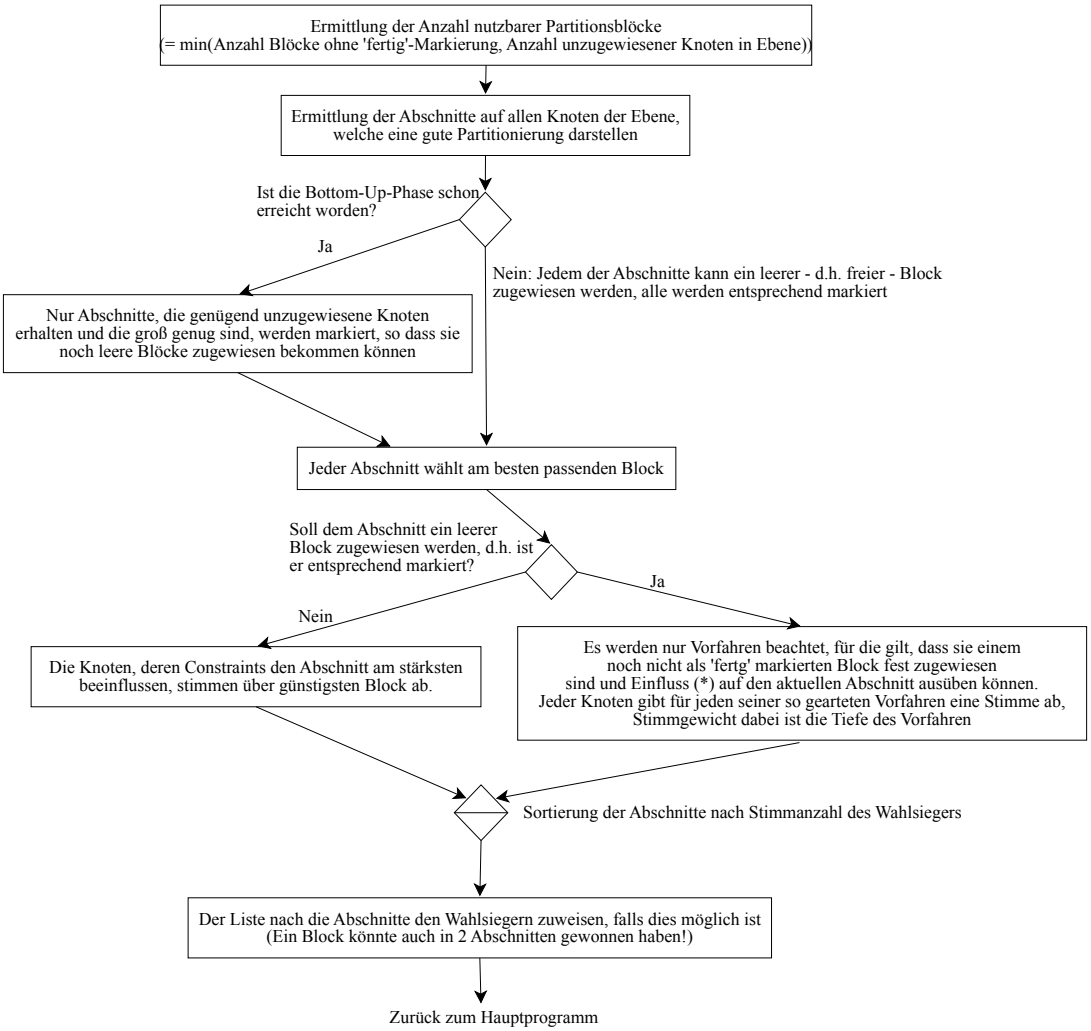

Abbildung B.5: Flussdiagramm für die 2. Runde der Entscheidungsphase, mit (*) gekennzeichnete Konzepte sind in Abschnitt 5.4.2 erklärt

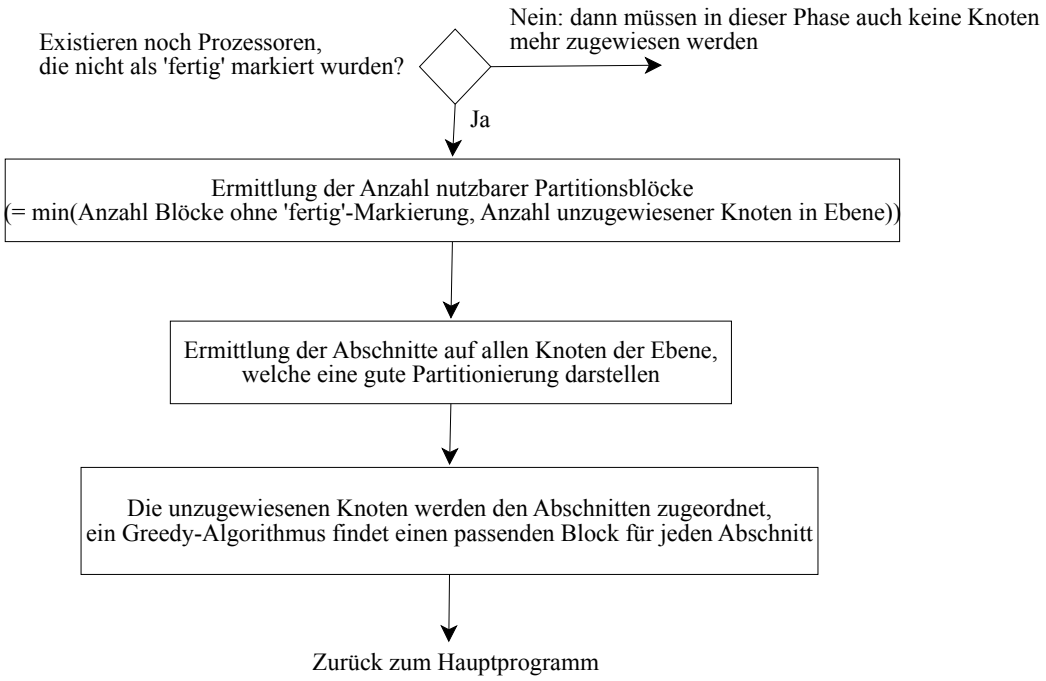

Abbildung B.6: Flussdiagramm für die 3. Runde der Entscheidungsphase

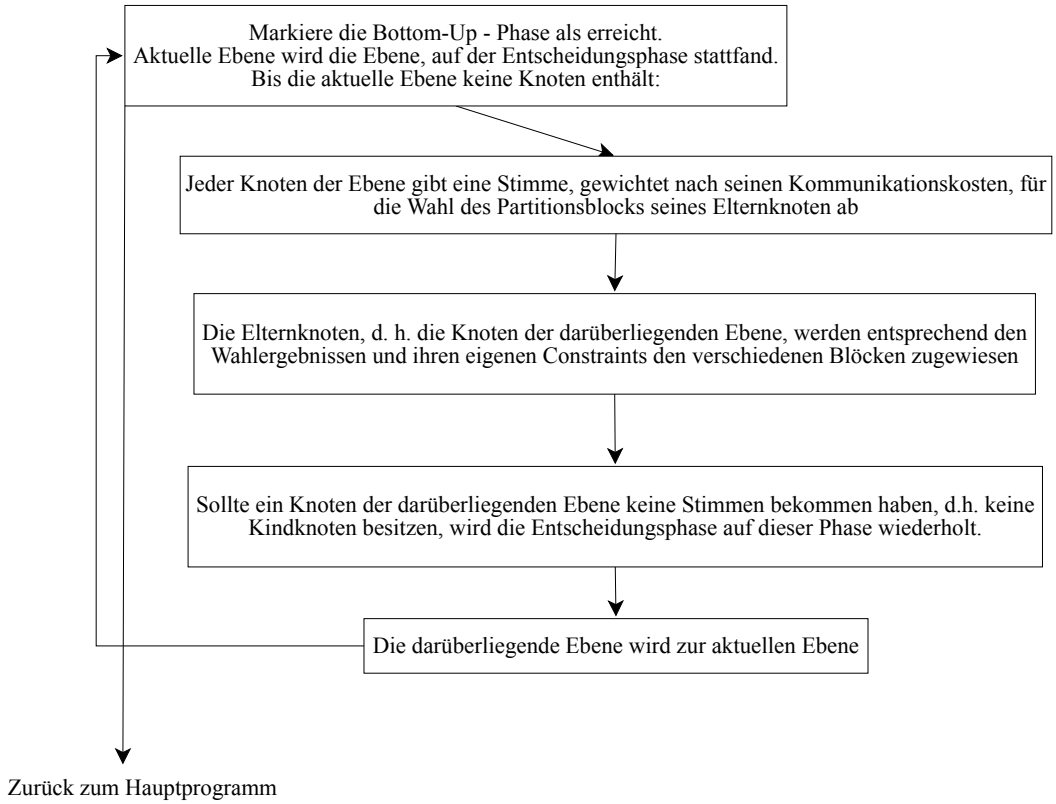

Abbildung B.7: Flussdiagramm für die Bottom-up - Phase

Anhang C

Weitere Tests des DEVS-Algorithmus

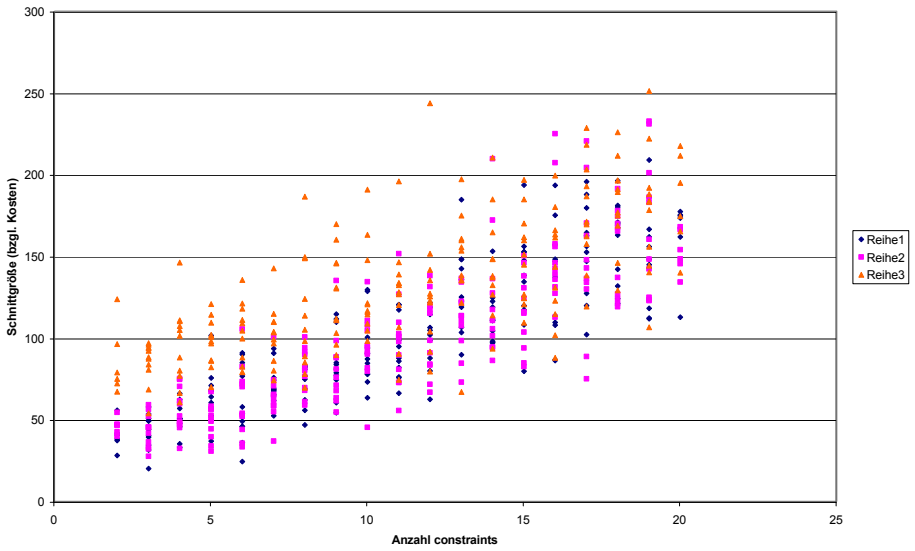

Abbildung C.1: Schnittgröße in Abhängigkeit von der Anzahl Constraints: Der Modellgraph bestand aus 200 Knoten. In Reihe 1 hat der Modellgraph den Verzweigungsfaktor 4, und er wird auf 8 Prozessoren aufgeteilt. In Reihe 2 gibt es wiederum 8 Prozessoren, aber diesmal hat der Baum den Verzweigungsfaktor 8. In Reihe 3 beträgt der Verzweigungsfaktor auch 8, aber es wird auf 16 Prozessoren aufgeteilt. Der Verzweigungsfaktor scheint die Größe des Schnittes nicht stark zu verändern, allerdings wird die Schnittgröße durch die Anzahl der zu verwendenden Prozessoren beeinflusst.

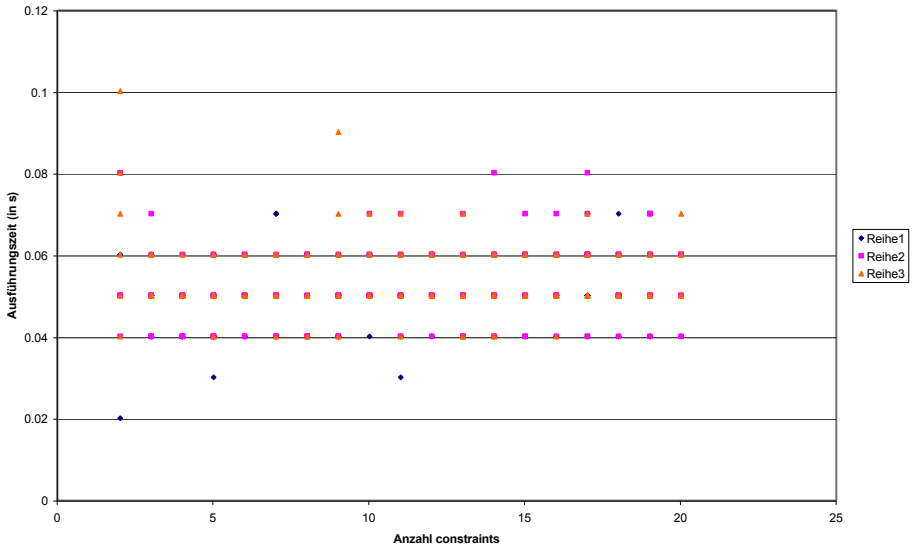

Abbildung C.2: Ausführungszeit in Abhängigkeit von der Anzahl Constraints: Verwendung der gleichen Testreihen wie bei Abbildung C.1 beschrieben. Die Ausführungszeit wird offensichtlich nicht durch die Anzahl der Constraints beeinflusst.

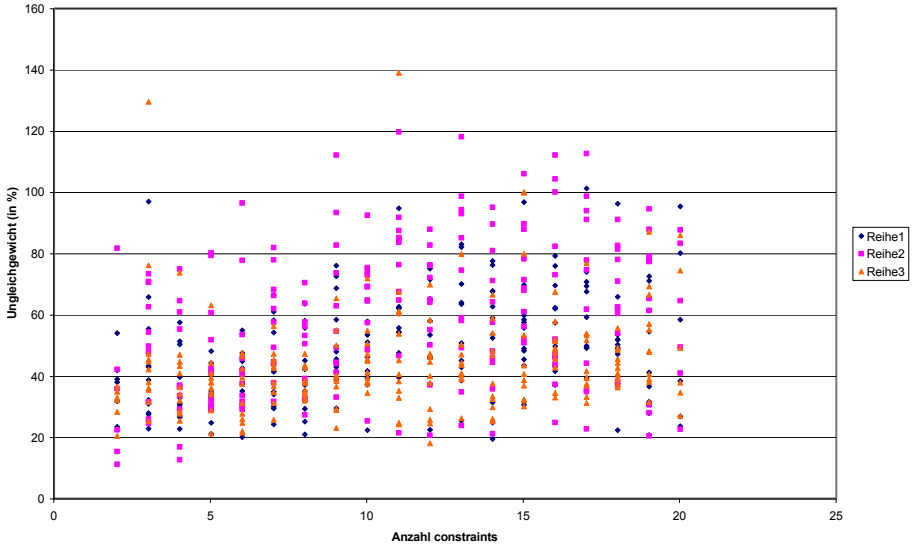

Abbildung C.3: Ungleichgewicht in Abhängigkeit von der Anzahl Constraints: Verwendung der gleichen Testreihen wie bei Abbildung C.1 beschrieben. Es ist zu erkennen, dass die Kombination aus Anzahl der Partitionsblöcke und dem Verzweigungsfaktor eine wichtige Rolle für das Gleichgewicht der Partitionsblöcke spielt (Reihe 2 besitzt breiter gestreute Werte als die anderen beiden Reihen). Außerdem wird deutlich, dass auch bei sehr wenigen Constraints extrem ungünstige Fälle auftreten können.

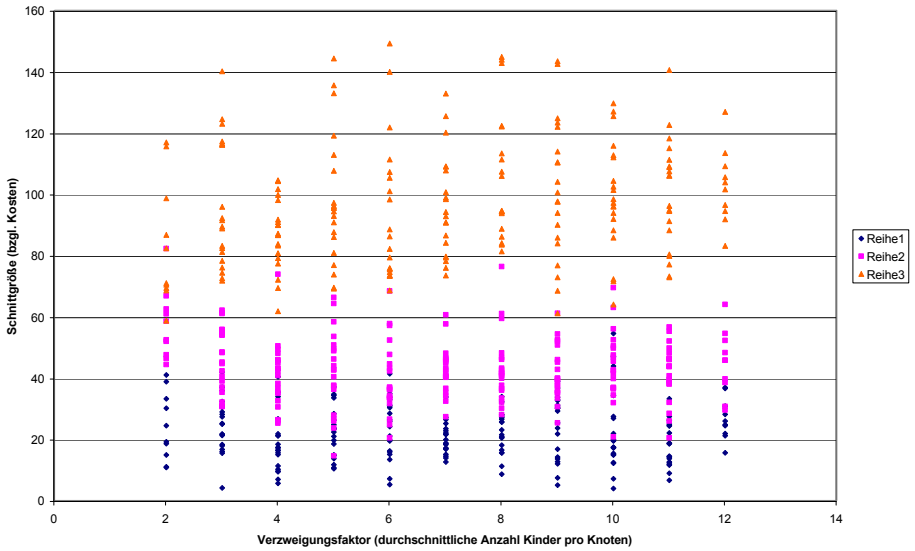

Abbildung C.4: Schnittgröße in Abhängigkeit vom Verzweigungsfaktor: Der Modellgraph bestand aus 200 Knoten, es wurden jeweils 2 Constraints festgelegt. Die Anzahl der Prozessoren wurde auf 4 (Reihe 1), 8 (Reihe 2) und 16 (Reihe 3) festgelegt. In diesem Diagramm ist gut zu erkennen, dass der Verzweigungsfaktor - im Gegensatz zur Anzahl der verwendeten Prozessoren - keinen Einfluss auf die Schnittgröße ausübt.

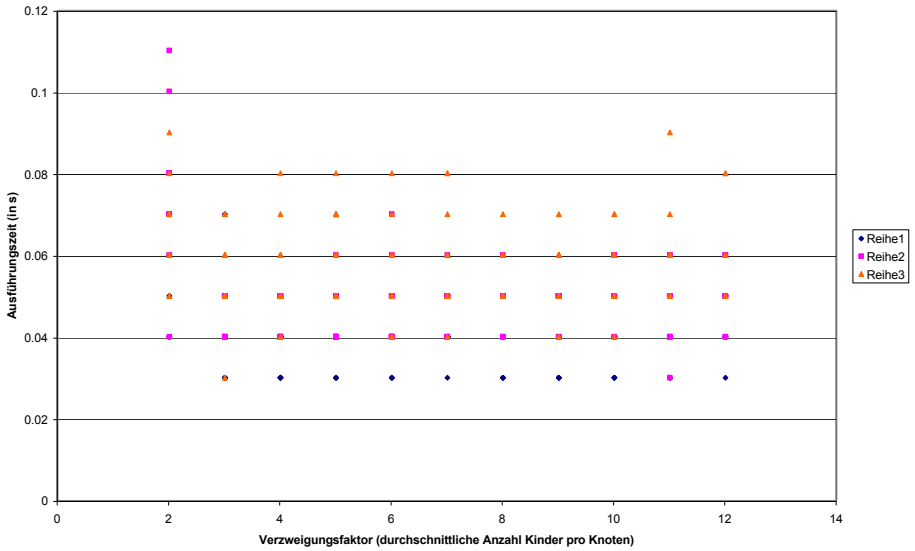

Abbildung C.5: Ausführungszeit in Abhängigkeit vom Verzweigungsfaktor: Verwendung der gleichen Testreihen wie bei Abbildung C.4 beschrieben. Die Ausführungszeit ist offensichtlich unabhängig vom Verzweigungsfaktor des Modellgraphen. Bei einem sehr kleinen Verzweigungsfaktor ist es jedoch schwieriger, eine geeignete Entscheidungsebene zu finden, daher benötigten die entsprechenden Läufe etwas mehr Zeit.

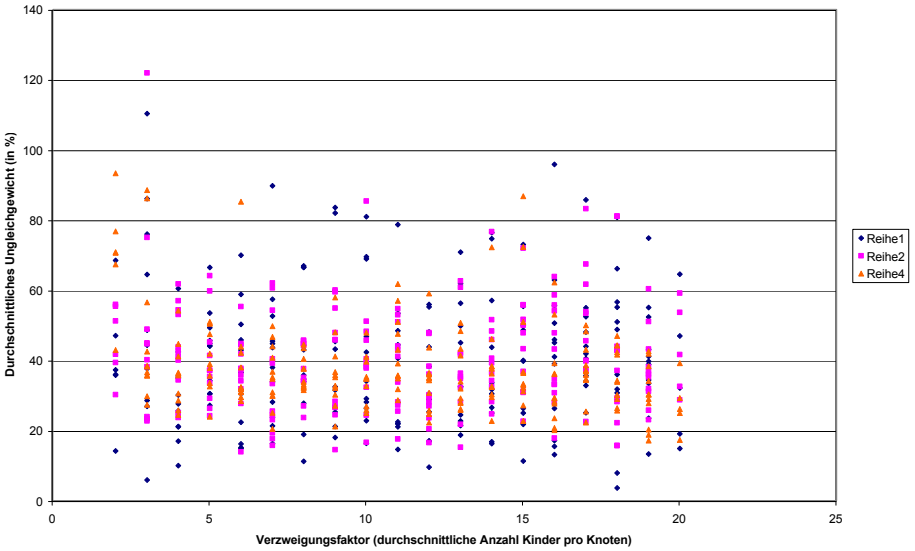

Abbildung C.6: Ungleichgewicht in Abhängigkeit vom Verzweigungsfaktor: Verwendung der gleichen Testreihen wie bei Abbildung C.4 beschrieben. Das erzielte Ungleichgewicht scheint weder von vom Verzweigungsfaktor noch von der Anzahl der Prozessoren abzuhängen - es sei denn, bei sehr kleinem Verzweigungsfaktor und ungünstig liegenden Constraints. Damit zeigt sich, dass die Sonderfall-Heuristiken des Algorithmus durchaus noch verbesserungswürdig sind.

Glossar

Ableitungsbaum

Der Ableitungsbaum eines Wortes aus einer kontextfreien Sprache ist die Darstellung der zur Produktion des Wortes genutzten Regeln als Baum. Jede Regelanwendung wird durch eine Kante zwischen dem Knoten mit der Symbolfolge, auf der diese Regel angewendet wurde, und einem (neuen) Knoten mit der Symbolfolge nach Regelanwendung symbolisiert. Siehe *Kontextfreie Grammatik*, *Attributierte Grammatik*, Abschnitt 3.4.2.

Agent

Agenten sind Softwarekomponenten, die in einem gewissen Rahmen autonom und proaktiv agieren können. Das Verhalten von Agenten wird meist mit KI-Techniken gesteuert, so dass Agenten sich eigene Teilziele setzen, planen oder auch mit anderen Agenten kommunizieren können. Viele Agenten besitzen zudem noch den Aspekt der Mobilität, das heißt sie können sich innerhalb einer definierten Umgebung, zum Beispiel einem Netzwerk, frei bewegen. Siehe *Multiagentensystem*.

Ausgabeport

Bestandteil eines DEVS-Modells. Verbindet die Ausgabe des Modells über eine Kopplung mit dem Eingabeport eines anderen Modells. Siehe *Eingabeport*, *Kopplung*.

Bandbreitenverringerung

Umordnung von Zeilen und Spalten einer Matrix, um alle Elemente, die ungleich 0 sind, näher an der Hauptdiagonale anzuordnen. Siehe Abschnitt 2.1.

Bisektion

Partition, die einen Graphen auf zwei verschiedene Partitionsblöcke aufteilt. Siehe Abschnitt 2.2.

Bottom-up

Bezeichnung für Verfahren, die in gewisser Weise von unten nach oben vorgehen. In dieser Arbeit wird der Begriff im Zusammenhang mit Bäumen verwendet, das heißt ein Bottom-up Verfahren auf einem Baum arbeitet von den Blättern ausgehend in Richtung der Wurzel. Gegenteil von *Top-down*.

Cluster

Ein Cluster ist ein Netz aus homogenen Computern. Die Rechner eines Clusters befinden sich meist nahe beieinander, um eine schnelle Netzwerkverbindung zwischen ihnen zu gewährleisten.

Constraint, konkret

Weist ein Modellelement einem konkreten Prozessor zu. Siehe Abschnitt 3.2.1.

Constraint, unbestimmt

Weist zwei Modellelemente dem gleichen - jedoch unbestimmten - Prozessor zu. Siehe Abschnitt 3.2.1.

Constraints

Unter Constraints (engl., 'Beschränkungen') versteht man die Einschränkung des Suchraums bei einem Suchproblem. Am einfachsten lassen sich Constraints als explizite, zusätzliche Bedingungen vorstellen, etwa 'Die gesuchte Zahl muss dreistellig sein.' oder 'Der gesuchte Graph darf höchstens 5 Kanten besitzen.'.

DEVS-Algorithmus

Damit ist der in Kapitel 5 beschriebene Partitionierungsalgorithmus für DEVS - Modelle gemeint.

Ebene

Mit einer Ebene ist die Menge aller Knoten eines Modellbaums gemeint, deren Pfadlänge zum Wurzelknoten gleich ist. Ist die Pfadlänge der Knoten einer Ebene kleiner, so wird die Ebene als *höher* bezeichnet (bezüglich der Graphenvisualisierung in welcher die Wurzel am Weitesten oben angeordnet ist), ansonsten als *tiefer*. Ebenen werden hier meist als Tupel betrachtet, das heißt jeder Knoten hat eine gewisse Position in der Ebene. Siehe *Entscheidungsebene*, *Tupel*, *Zerlegungspunkt*, Kapitel 5.

Eingabeport

Bestandteil eines DEVS-Modells. Erlaubt es dem Modell, die Ausgaben eines anderen Modells über eine Kopplung entgegenzunehmen. Siehe *Ausgabeport*, *Kopplung*.

Entscheidungsebene

Die in Kapitel 5 beschriebene Entscheidungsebene ist die Ebene des Modellbaums, die hauptsächlich für seine Partitionierung verwendet wird. Siehe *Ebene*, Kapitel 5.

Framework

Ein Framework ist zunächst ein Sammlung von Komponenten, die zur Lösung bestimmter Aufgaben eingesetzt werden können. Anders als normale Software - Bibliotheken besitzen Frameworks meist nur sehr wenige Funktionen, die der Nutzer aufrufen kann. Dafür sind sie meist schneller

und einfacher erweiterbar, das besondere Augenmerk liegt hier also auf der Schaffung von Rahmenbedingungen und Referenz-Implementierungen zur Lösung eines Problems.

Genetische Algorithmen

Eine Optimierungstechnik, die Erkenntnisse aus der Evolutionstheorie zur Durchsuchung des Suchraums verwendet. Siehe Abschnitt 2.4.

Grammatik

Eine Grammatik ist eine Menge von (Produktions-)Regeln, mit denen man Wörter einer formale Sprache produzieren kann. Daher definiert eine Grammatik eine formale Sprache.

Grammatik, attributiert

Eine attributierte Grammatik ist eine kontextfreie Grammatik, bei der jedes Nichtterminalsymbol zusätzlich noch eine Menge von Attributen besitzen darf. Der Wert eines Attributs lässt sich bei Ausführung einer Regel berechnen. Im Ableitungsbaum eines Wortes einer attributierten Grammatik erfolgt die Berechnung der synthetisierten Attribute von unten nach oben, und die der vererbten Attribute von oben nach unten. Siehe *Kontextfreie Grammatik*, *Ableitungsbaum*, Abschnitt 3.4.2.

Grammatik, kontextfrei

Bei einer kontextfreien Grammatik dürfen die linken Seiten der Produktionsregeln nur aus einem Nichtterminal bestehen. Nichtterminale sind Symbole, die von Produktionsregeln in Folgen von Terminal- und Nichtterminalsymbolen umgewandelt werden müssen. Die Terminalsymbole bilden nach der Ausführung der Produktionsregeln das Wort der formalen Sprache. Anders ausgedrückt spielt der Kontext eines Nichtterminals (andere Symbole in seiner Umgebung) bei den Produktionsregeln einer kontextfreien Grammatik keine Rolle. Durch kontextfreie Grammatiken definierbare Sprachen heißen *kontextfreie Sprachen*. Eine genaue Definition ist in [27] nachzulesen.

Graph, komplett

Ein Graph, bei dem jeder Knoten mit jedem anderen Knoten genau eine Kante besitzt.

Graph, planar

Graphen, die man auf einer zweidimensionalen Ebene zeichnen kann, ohne dass sich zwei Kanten überschneiden.

Graph, schlicht

Ein Graph ohne Schlingen (Kanten von einem Knoten zu sich selbst) und Mehrfachkanten.

Greedy - Algorithmus

Grob gesagt sind Greedy-Algorithmen Verfahren, welche die Teillösung eines Problems durch schrittweises Hinzufügen einer lokal 'besten' weiteren Teillösung zu einer Gesamtlösung erweitern. Näheres zu Greedy-Algorithmen findet sich in [24].

Grid

Ein Grid ist ein Menge von durch eine zusätzliche *Grid*-Software verwalteten Computern. Im Gegensatz zu *Clustern* besteht ein Grid meist aus sehr verschiedenen Komponenten, die weit voneinander entfernt sein können.

Hillclimbing

Optimierungsverfahren, bei dem die Parameter einer Funktion schrittweise in die aussichtsreichste Richtung geändert werden. Bildlich gesprochen wird die Schrittrichtung immer 'bergauf' gewählt, und zwar so, dass der Wert der Funktion nach jedem Schritt, also jeder Parameteränderung, mindestens einen gleich hohen Wert besitzt wie im vorigen Schritt. Dadurch findet man mit diesem Verfahren meist nur lokale Maxima. Dieses Verfahren lässt sich beispielsweise durch *Simulated Annealing* verfeinern.

Infrastrukturgraph

Ein Graph, bei dem die Knoten die verfügbaren Prozessoren in einem Rechnerverbund symbolisieren. Kanten zwischen den Knoten deuten eine existente Kommunikationsmöglichkeit zwischen den beiden symbolisierten Prozessoren an.

Java Virtual Machine (JVM)

Die Java Virtual Machine ist die Spezifikation eines Java-Bytecode - Interpreters. Dieser startet Java-Bytecode innerhalb einer so genannten virtuellen Maschine, auf deren Basis der Bytecode interpretiert wird. Durch die Verfügbarkeit von JVMs für eine Vielzahl von Plattformen gilt Java als plattformunabhängig. Neben der Referenzimplementierung von Sun [38] gibt es noch zahlreiche Implementierungen von anderen Herstellern, die ihre Interpreter für gewisse Plattformen und Aufgabenbereiche optimiert haben.

Koordinator, (DEVS-)

Ein (DEVS-)Koordinator ist Teil einer DEVS-Simulation. Er wird mit einem gekoppelten Modell assoziiert. Siehe *Simulator*, Abschnitt 1.3.

Kopplung

Verbindet die Ein- und Ausgabeports zweier Modelle miteinander. Siehe *Eingabeport*, *Ausgabeport*, Abschnitt 1.3.

Load Balancing

Im Gegensatz zur Modellpartitionierung, bei der eine *initiale* Verteilung des Modells berechnet wird, geht es beim Load Balancing um die Erhaltung einer möglichst guten Verteilung zur Laufzeit, die gegebenenfalls durch *Repartitionierung* erreicht wird.

Modell, (DEVS-)

Ein DEVS-Modell (Abk. engl., Diskret-Ereignisorientiertes System) ist ein ereignisorientiertes Modell, dass mit Hilfe des DEVS-Modellierungsformalismus spezifiziert wurde. Siehe Abschnitt 1.3.

Modell, atomar

Atomare (DEVS-)Modelle bilden die kleinste Einheit eines DEVS-Modells. Sie besitzen unter anderem einen internen Zustand, sowie Funktionen zur Modellierung von Zustandsübergängen verschiedener Art. Siehe *gekoppeltes Modell*, Abschnitt 1.3.

Modell, gekoppelt

Ein gekoppeltes (DEVS -) Modell ist ein aus atomaren oder gekoppelten Modellen bestehendes Modell, dessen Verhalten durch diese Bestandteile und deren Kommunikation untereinander definiert wird. Nach außen hin verhält es sich ebenfalls wie ein atomares Modell. Siehe *atomares Modell*, Abschnitt 1.3.

Modellbaum

Ein Modellgraph, der die Baumeigenschaft besitzt, das heißt er besitzt einen Wurzelknoten und hat keine Kreise. Siehe *Modellgraph*.

Modellgraph

Ein Graph, bei dem die Knoten die Elemente eines Modells symbolisieren. Kanten zwischen den Knoten deuten eine mögliche Interaktion zwischen den beiden symbolisierten Komponenten des Modells an. Siehe *Infrastrukturgraph*.

Modellierungsformalismus

Ein Modellierungsformalismus ist eine Menge an Regeln, Formeln und Vereinbarungen, welche die formale Spezifikation eines Modells erleichtern. Die meisten Modellierungsformalismen eignen sich besonders für Modelle mit speziellen Eigenschaften. In dieser Arbeit werden *zelluläre Automaten*, *DEVS-Modelle* und *StateCharts* als Beispiele für Modellierungsformalismen verwendet.

Multiagentensystem

System, dessen Elemente zum Teil Agenten darstellen. Diese können meist miteinander interagieren. Durch die Verwendung von KI-Techniken in den Agenten selbst, sowie die Erhöhung der Komplexität durch die Interaktion zwischen ihnen, sind Multiagentensysteme meist sehr dynamisch. Siehe *Agent*.

Pfadlänge

Die Pfadlänge zwischen zwei Knoten eines Graphen wird in dieser Arbeit als die Anzahl an Kanten angesehen, die im kürzesten Weg zwischen den zwei Knoten im Graphen liegen.

Polymorphie

Polymorphie (gr., Vielgestaltigkeit) ist unter anderem ein Kernkonzept objektorientierter Programmiersprachen, dass es dem Programmierer erlaubt, einem Objekt mehrere Typen zuzuordnen. In Java ist dies mit Hilfe von Vererbung oder der Implementierung von Schnittstellen möglich. Grob gesagt bietet Polymorphie den Vorteil, dass Objekte im Programmcode

verarbeitet werden können, deren genauer Typ erst zur Laufzeit festgestellt werden kann.

PPP

Das Point-to-Point Protocol (PPP) ist ein Netzwerkprotokoll, dass zum Beispiel zur Einwahl ins Internet über ein Modem oder eine ISDN-Karte verwendet wird.

Repartitionierung

Wiederholte Partitionierung eines Modells, im Allgemeinen mit Rückgriff auf eine bestehende Partition. Siehe *Load Balancing*.

schwach besetzte Matrix

Eine Matrix, bei der eine große Mehrheit der Elemente den Wert 0 hat.

Simulated Annealing

Eine durch die Metallurgie inspirierte Optimierungsmethode, die im Wesentlichen auf dem *Hillclimbing* - Verfahren aufsetzt. Im Unterschied dazu wird ein Schritt beim Simulated Annealing (engl., simuliertes Hartglühen) mit einer gewissen Wahrscheinlichkeit auch in weniger aussichtsreiche Regionen des Suchraums ausgeführt. Die Wahrscheinlichkeit für dieses 'bergab'-Wandern wird dann schrittweise verringert. Siehe *Hillclimbing*, [13].

Simulator, (DEVS-)

Ein DEVS-Simulator wird mit einem atomaren DEVS-Modell assoziiert und simuliert dieses während der Simulation. Siehe *Koordinator*, Abschnitt 1.3.

Simulator-Objekt

Ein Objekt im JAMES II - System, welches ein gegebenes Modell simuliert. Die Modelle und die Logik für deren Ausführung sind bei JAMES II getrennt.

Software Pattern

Entwurfsmuster, die allgemeine Konzepte beim Softwareentwurf veranschaulichen sollen. Werden speziell für den Entwurf objektorientierter Softwaresysteme verwendet.

sparse Matrix

Siehe *schwach besetzte Matrix*.

Sprache, kontextfrei

Siehe *Kontextfreie Grammatik*.

StateCharts

StateCharts (engl., Zustandsdiagramme) sind ein Modellierungsformalismus innerhalb von UML, mit dem zustandsbasierte, ereignisorientierte Systeme modelliert werden können. Siehe *UML*.

Swapping

Unter Swapping (engl., das Austauschen) versteht man im Kontext der Betriebssysteme das Auslagern von Daten, die auf Grund ihrer Größe nicht mehr im Arbeitsspeicher gehalten werden können, in den um mehrere Größenordnungen langsameren Festplattenspeicher.

Top-down

Bezeichnung für Verfahren, die in gewisser Weise von oben nach unten vorgehen. In dieser Arbeit wird der Begriff im Zusammenhang mit Bäumen verwendet, das heißt ein Top-down Verfahren auf einem Baum arbeitet von der Wurzel ausgehend in Richtung der Blätter. Gegenteil von *Bottom-up*.

Tupel

Ein Tupel ist das mathematische Konzept einer geordnete Menge von Elementen, am besten vergleichbar mit einer Liste.

UML

UML (Unified Modeling Language - engl., Vereinheitlichte Modellierungssprache) ist eine Menge meist graphischer Modellierungsformalismen, die vor allem zur Spezifikation von Softwaresystemen verwendet werden. Gebräuchliche Formalismen sind dabei Klassendiagramme zur Darstellung von Strukturen in objektorientierten Sprachen sowie StateCharts, mit denen das Verhalten der Instanzen von Klassen modelliert werden kann. Näheres dazu ist unter [41] zu finden. Siehe *StateCharts*.

Ungleichgewicht

Maß für die Abweichung einer Partition von einer gerechten Aufteilung bezüglich Rechenbedarf und Rechenkapazität. Siehe Abschnitt 3.5.

Verzweigungsfaktor

Bei gewöhnlichen Graphen ist der Verzweigungsfaktor die durchschnittliche Anzahl der Nachbarn, bei Bäumen ist damit die durchschnittliche Anzahl an Kindern pro Knoten gemeint. Das heißt Bäume mit Verzweigungsfaktor k hätten als normale Graphen einen Verzweigungsfaktor von $k+1$, wegen der Kanten zu den Elternknoten. Wird zur Parametrisierung der Erzeugung von zufälligen Graphen und Bäumen verwendet. Siehe Abschnitt 3.5 und Kapitel 6.

Zelluläre Automaten

Zelluläre Automaten sind ein Modellierungsformalismus, bei dem die Menge der vorhandenen Modelle homogen ist, und die Modelle in einem Gitternetz angeordnet werden. Der Zustand eines Modellelements ist dabei abhängig von den Zuständen seiner Nachbarelemente im Gitter.

Zerlegungspunkt

Markiert eine Stelle innerhalb einer Ebene, welche zwei Zonen voneinander abgrenzt. Siehe *Zone*, *Ebene*, Abschnitt 5.4.2.

Zone

Gewisse Menge von nebeneinander liegenden, unzugewiesenen Knoten innerhalb einer Ebene. Dieser Begriff wird ausschließlich zur einfacheren Beschreibung des DEVS-Partitionierungsalgorithmus verwendet. Siehe *Zerlegungspunkt*, Abschnitt 5.4.2.

Abbildungsverzeichnis

2.1 Einordnung der vorgestellten Partitionierungsverfahren 12
2.2 Skizze zur Grundidee der rekursiven spektralen Bisektion 16
2.3 Beispiele für Automorphismen von Graphen 27
2.4 Beispiele für die Zerlegung eines Graphen nach Orbits 28
2.5 Illustration weiterer Begriffe 30

3.1 Anwendungsbeispiel für Constraints 37
3.2 Vereinfachte Struktur des Hauptpakets 43
3.3 Aufbau des Frameworks für Partitionierungsalgorithmen 43
3.4 Beispiele für synthetisierte und vererbte Attribute 46

4.1 Klassendiagramm der Graphenklassen 55
4.2 Beispiel für die Darstellung eines Partitionierungsergebnisses . . 56

5.1 Flussdiagramm für generellen Ablauf 59
5.2 Intuitive Einteilung eines Modellbaumes 60
5.3 Distanz zweier Geschwisterknoten 62
5.4 Allgemeine Distanz zweier Knoten einer Ebene 62
5.5 Beispiel für Constraint-Konflikt in Top-down Phase 63
5.6 Wichtigkeit der Wahl der Entscheidungsebene 64
5.7 Schwieriger Fall für die Wahl der Entscheidungsebene 64
5.8 Flussdiagramm für die Entscheidungsphase 65
5.9 Flussdiagramm für die 1. Runde der Entscheidungsphase 67
5.10 Motivation für Blockwahl in der 2. Runde 68
5.11 Allgemeine Heuristik zur Partitionierung 69
5.12 Ermittlung der Knotenmengen zur Blockwahl in Runde 2 69
5.13 In Runde 2 der Entscheidungsphase beachtete Constraints 70
5.14 Entscheidung über Zuweisung von Knoten in neuen Block 71
5.15 Fallunterscheidung für die Blockwahl in Zonen 72
5.16 Flussdiagramm für die 2. Runde der Entscheidungsphase, (*): Mit Einfluss ist hier die in Abbildung 5.13 beschriebene Eigenschaft gemeint 73
5.17 Veranschaulichung der Bottom-up - Phase 75
5.18 Beispiel für einen ungünstigen Modellbaum 77

6.1 Einfluss von Parametern des DEVS-Algorithmus 1 81
6.2 Einfluss von Parametern des DEVS-Algorithmus 2 82
6.3 Schnittgröße in Abhängigkeit von der Größe des Modellgraphen . 83

6.4 Ausführungszeit in Abhängigkeit von der Größe des Modellgraphen 84
6.5 Ungleichgewicht in Abhängigkeit von der Größe des Modellgraphen 85
6.6 Ausführungszeiten des DEVS-Algorithmus bei besonders großen Modellgraphen 86
6.7 Vergleich zwischen Standard- und DEVS-Algorithmus 87

A.1 Klassendiagramm Partitionierungsalgorithmen 92
A.2 Klassendiagramm Modellanalysealgorithmen 93
A.3 Klassendiagramm Infrastrukturanalysealgorithmen 93
A.4 Vereinfachtes Klassendiagramm für Partitionizer - Paket 93
A.5 Klassendiagramm für DEVS-Partitionierungsalgorithmus 94
A.6 Klassendiagramm für Graphen-Klassen 94

B.1 Flussdiagramm für die Abfolge der Phasen 95
B.2 Flussdiagramm für die Top-down - Phase 96
B.3 Flussdiagramm für die Entscheidungsphase 96
B.4 Flussdiagramm für die 1. Runde der Entscheidungsphase 97
B.5 Flussdiagramm für die 2. Runde der Entscheidungsphase 98
B.6 Flussdiagramm für die 3. Runde der Entscheidungsphase 99
B.7 Flussdiagramm für die Bottom-up - Phase 100

C.1 Schnittgröße in Abhängigkeit von der Anzahl Constraints 101
C.2 Ausführungszeit in Abhängigkeit von der Anzahl Constraints . . 102
C.3 Ungleichgewicht in Abhängigkeit von der Anzahl Constraints . . 102
C.4 Schnittgröße in Abhängigkeit vom Verzweigungsfaktor 103
C.5 Ausführungszeit in Abhängigkeit vom Verzweigungsfaktor 104
C.6 Ungleichgewicht in Abhängigkeit vom Verzweigungsfaktor 105

Tabellenverzeichnis

4.1 Semantik der Graphenbeschriftung 54

5.1 Parameter des DEVS-Partitionierungsalgorithmus 78

6.1 Testergebnisse des Standardalgorithmus 86

Literaturverzeichnis

[1] P.-O. Fjällström. Algorithms for Graph Partitioning: A Survey, Linköping Articles in Computer and Information Science Vol. 3, 1998

[2] M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-completeness. W. H. Freeman, 1979

[3] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs, Bell System Tech. Journal, Vol. 49, 1970

[4] C. M. Fiduccia, R. M. Mattheyses. A linear-time heuristic for improving network partitions, Proceedings of the 19th conference on Design automation, 1982

[5] B. Hendrickson, R. Leland. A multilevel algorithm for partitioning graphs. Proceedings of the IEEE/ACM SC95 Supercomputing Conference, 1995

[6] K. Schloegel, G. Karypis, V. Kumar. Graph Partitioning for High Performance Scientifc Simulations. Draft for CRPC Parallel Computing Handbook, Morgan Kaufman, 2000

[7] A. Pothen, H. D. Simon, K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. SIAM Journal on Matrix Analysis and Applications, 1990

[8] H. D. Simon. Partitioning of Unstructured Problems for Parallel Processing. Computing Systems in Engineering, 1991

[9] T. M. J. Fruchterman, E. M. Reingold. Graph drawing by force-directed placement. Software—Practice & Experience Vol. 21, 1991

[10] H.S. Maini, K. G. Mehrotra, C. K. Mohan, S. Ranka. Genetic Algorithms for Graph Partitioning and Incremental Graph Partitioning. Center for Research on Parallel Computation, 1994

[11] J. H. Holland. Adaptation in natural and artificial systems. Ann Arbor: The University of Michigan Press, 1975

[12] A. E. Langham, P. W. Grant. Using Competing Ant Colonies to Solve k-way Partitioning Problems with Foraging and Raiding Strategies. Proceedings of the 5th European Conference on Advances in Artificial Life, 1999

[13] A. Chatterjee, R. Hartley. A new simultaneous circuit partitioning and chip placement approach based on simulated annealing. Proceedings of the 27th ACM/IEEE conference on Design automation, 1991

[14] S. Park, B. P. Zeigler. Distributing Simulation Work Based on Component Activity: A New Approach to Partitioning Hierarchical DEVS Models. Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments, IEEE 2003

[15] S. Park, C. Hunt, B. P. Zeigler. Generic Model Paritioning with LookAhead k: A Multi-Scale Partitioning Algorithm for DEVS Biomimetic In Silicio Devices, pre-print Version for Agent-Directed Simulation Symposium 2005

[16] J. Lemeire, B. Smets, Ph. Cara, E. Dirkx. Exploiting Symmetry for Partitioning Models in Parallel Discrete Event Simulation, Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS), IEEE 2004

[17] P Foggia, C. Sansone, M. Vento. A Performance Comparison of Five Algorithms for Graph Isomorphism, Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representation, 2001

[18] D. P. Spooner, S. A. Jarvis, J. Cao, S. Saini, G. R. Nudd. Local Grid Scheduling Techniques using Performance Prediction, IEE Proceedings: Computers and Digital Techniques, 2003

[19] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, 1995

[20] J. W. Cooper. Java ^{TM}Design Patterns: A Tutorial, Addison Wesley, 2000

[21] L. Li, H. Huang, C. Topper. DVS: An Object-Oriented Framework for Distributed Verilog Simulation. Proceedings of the 17th Workshop on Parallel and Distributed Simulation (PADS), IEEE 2003

[22] D. E. Knuth. On the Translation of Languages from Left to Right. Information and Control 8, 1965

[23] R. Kickuth. Petaflop/s für Proteine, CLB Chemie in Labor und Biotechnik 11, 2002

[24] A. Brandstädt. Graphen und Algorithmen, Teubner Verlag, 1996

[25] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices, Proc. 24th Nat. Conf. ACM, 1969.

[26] D. Lau. Algebra und diskrete Mathematik, Band 2. Springer-Verlag, 2004

[27] K. Erk, L. Priese. Theoretische Informatik. Teubner-Verlag, 2001

[28] M. R. Garey, D. S. Johnson, L. Stockmeyer. Some simplified NP-complete problems, STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing, 1974

[29] B.P. Zeigler. DEVS representation of dynamical systems: event-based intelligent control, Proceedings of the IEEE Vol.77, 1989

[30] A. Singla, T. M. Conte. Bipartitioning for hybrid FPGA-software simulation. 9th International Conference on VLSI Design, 1996

[31] http://cs.anu.edu.au/people/bdm/nauty/

[32] http://folding.stanford.edu/

[33] http://www.research.ibm.com/bluegene/

[34] http://staffweb.cms.gre.ac.uk/ c.walshaw/partition/

[35] http://www.math.uib.no/ bjornoh/mtj/

[36] http://www.graphviz.org/

[37] http://www.eclipse.org/

[38] http://java.sun.com/

[39] http://www.gentleware.com/

[40] http://www.gnome.org/projects/dia/

[41] http://www.uml.org/

Die Umsetzung der beschriebenen Lösung wurde mit der Entwicklungsumgebung *Eclipse 3.1* [37] unter *Java 1.5 (Standard Edition)* [38] vorgenommen.

Die Visualisierung der Graphen erfolgte mit dem Programm `dot` aus dem Graphenvisualisierungs-Toolkit *Graphviz*[36]. Die UML-Diagramme wurden mit dem UML-Editor *PoseidonUML* [39] erzeugt, die Flussdiagramme mit *dia* [40]. Die restlichen Abbildungen wurden mit herkömmlichen Bilbearbeitungswerkzeugen erstellt.